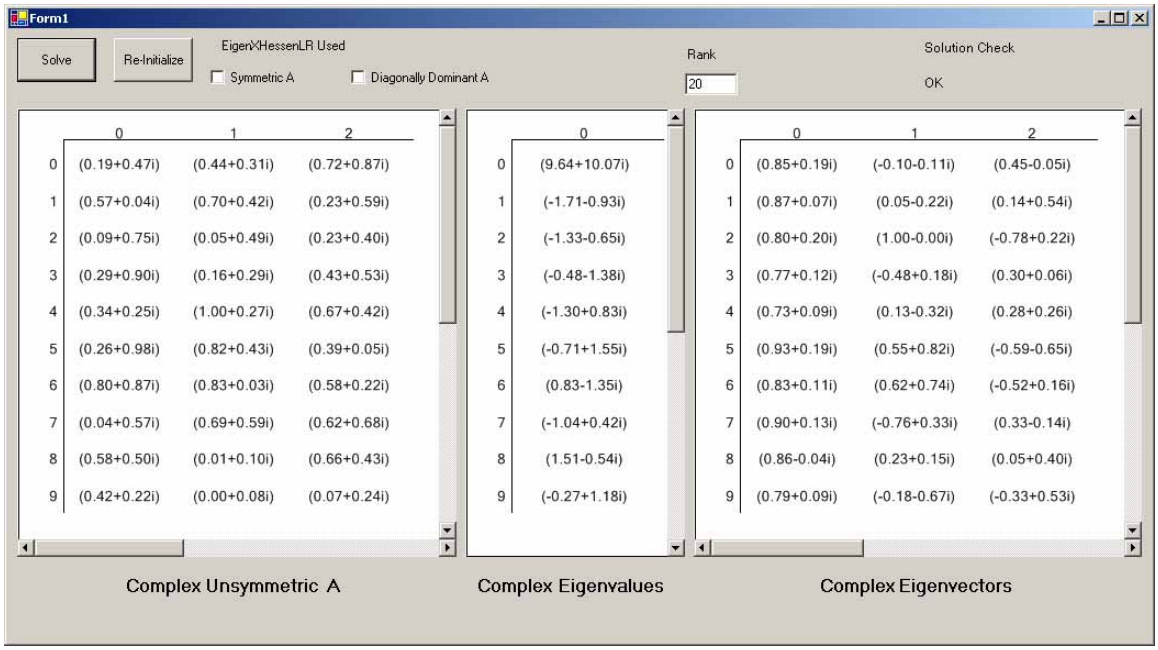
QCMatPack Matrix and Numerical Analysis Software for Java



Contact Information

Company Web Site: **http://**www.quinn-curtis.com

Electronic mail

General Information: info@quinn-curtis.com
Sales: sales@quinn-curtis.com

Technical Support Forum

http://www.quinn-curtis.com/ForumFrame.htm

Revision Date 11/24/2007 Rev. 1.0

QCMatPack for Java Documentation and Software Copyright **Quinn-Curtis, Inc.** 2007

Quinn-Curtis, Inc. QCMatPack Tools Java END-USER LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: This Software End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Quinn-Curtis, Inc. for the Quinn-Curtis, Inc. SOFTWARE identified above, which includes the Quinn-Curtis, Inc. **QCMatPack Matrix and Numerical Analysis Software.** By installing, copying, or otherwise using the SOFTWARE, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE. If the SOFTWARE was mailed to you, return the media envelope, UNOPENED, along with the rest of the package to the location where you obtained it within 30 days from purchase.

1. The SOFTWARE is licensed, not sold.

2. GRANT OF LICENSE.

(A) Developer License. After you have purchased the license for SOFTWARE, and have received the file containing the licensed copy, you are licensed to copy the SOFTWARE only into the memory of the number of computers corresponding to the number of licenses purchased. The primary user of the computer on which each licensed copy of the SOFTWARE is installed may make a second copy for his or her exclusive use on a portable computer. Under no other circumstances may the SOFTWARE be operated at the same time on more than the number of computers for which you have paid a separate license fee. You may not duplicate the SOFTWARE in whole or in part, except that you may make one copy of the SOFTWARE for backup or archival purposes. You may terminate this license at any time by destroying the original and all copies of the SOFTWARE in whatever form.

(B) 30-Day Trial License. You may download and use the SOFTWARE without charge on an evaluation basis for thirty (30) days from the day that you DOWNLOAD the trial version of the SOFTWARE. The termination date of the trial SOFTWARE is embedded in the downloaded SOFTWARE and cannot be changed. You must pay the license fee for a Developer License of the SOFTWARE to continue to use the SOFTWARE after the thirty (30) days. If you continue to use the SOFTWARE after the thirty (30) days without paying the license fee you will be using the SOFTWARE on an unlicensed basis.

Redistribution of 30-Day Trial Copy. Bear in mind that the 30-Day Trial version of the SOFTWARE becomes invalid 30-days after downloaded from our web site, or one of our sponsor's web sites. If you wish to redistribute the 30-day trial version of the SOFTWARE you should arrange to have it redistributed directly from our web site If you are using SOFTWARE on an evaluation basis you may make copies of the evaluation SOFTWARE as you wish; give exact copies of the original evaluation SOFTWARE to anyone; and distribute the evaluation SOFTWARE in its unmodified form via electronic means (Internet, BBS's, Shareware distribution libraries, CD-ROMs, etc.). You may not charge any fee for the copy or use of the evaluation SOFTWARE itself. You must not represent in any way that you are selling the SOFTWARE itself. You must distribute a copy of this EULA with any copy of the SOFTWARE and anyone to whom you distribute the SOFTWARE is subject to this EULA.

(C) Redistributable License. The standard Developer License permits the programmer to deploy and/or distribute applications that use the Quinn-Curtis SOFTWARE, royalty free. We cannot allow developers to use this SOFTWARE to create a graphics toolkit (a library or any type of graphics component that will be used in combination with a program development environment) for resale to other developers.

If you utilize the SOFTWARE in an application program, or in a web site deployment, should we ask, you must supply Quinn-Curtis, Inc. with the name of the application program and/or the URL where the SOFTWARE is installed and being used.

3. RESTRICTIONS. You may not reverse engineer, de-compile, or disassemble the SOFTWARE, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this

limitation. You may not rent, lease, or lend the SOFTWARE. You may not use the SOFTWARE to perform any illegal purpose.

4. SUPPORT SERVICES. Quinn-Curtis, Inc. may provide you with support services related to the SOFTWARE. Use of Support Services is governed by the Quinn-Curtis, Inc. polices and programs described in the user manual, in online documentation, and/or other Quinn-Curtis, Inc.-provided materials, as they may be modified from time to time. Any supplemental SOFTWARE code provided to you as part of the Support Services shall be considered part of the SOFTWARE and subject to the terms and conditions of this EULA. With respect to technical information you provide to Quinn-Curtis, Inc. as part of the Support Services, Quinn-Curtis, Inc. may use such information for its business purposes, including for product support and development. Quinn-Curtis, Inc. will not utilize such technical information in a form that personally identifies you.

5. TERMINATION. Without prejudice to any other rights, Quinn-Curtis, Inc. may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE.

6. COPYRIGHT. The SOFTWARE is protected by United States copyright law and international treaty provisions. You acknowledge that no title to the intellectual property in the SOFTWARE is transferred to you. You further acknowledge that title and full ownership rights to the SOFTWARE will remain the exclusive property of Quinn-Curtis, Inc. and you will not acquire any rights to the SOFTWARE except as expressly set forth in this license. You agree that any copies of the SOFTWARE will contain the same proprietary notices which appear on and in the SOFTWARE.

7. EXPORT RESTRICTIONS. You agree that you will not export or re-export the SOFTWARE to any country, person, entity, or end user subject to U.S.A. export restrictions. Restricted countries currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria. You warrant and represent that neither the U.S.A. Bureau of Export Administration nor any other federal agency has suspended, revoked or denied your export privileges.

8. NO WARRANTIES. Quinn-Curtis, Inc. expressly disclaims any warranty for the SOFTWARE. THE SOFTWARE AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE REMAINS WITH YOU.

9. LIMITATION OF LIABILITY. IN NO EVENT SHALL QUINN-CURTIS, INC. OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES OF ANY KIND ARISING OUT OF THE DELIVERY, PERFORMANCE, OR USE OF THE SUCH DAMAGES. IN ANY EVENT, QUINN-CURTIS'S LIABILITY FOR ANY CLAIM, WHETHER IN CONTRACT, TORT, OR ANY OTHER THEORY OF LIABILITY WILL NOT EXCEED THE GREATER OF U.S. $1.00 OR LICENSE FEE PAID BY YOU.

10. U.S. GOVERNMENT RESTRICTED RIGHTS. The SOFTWARE is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer SOFTWARE clause of DFARS 252.227-7013 or subparagraphs (c)(i) and (2) of the Commercial Computer SOFTWARE- Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is: Quinn-Curtis, Inc., 18 Hearthstone Dr., Medfield MA 02052 USA.

11. MISCELLANEOUS. If you acquired the SOFTWARE in the United States, this EULA is governed by the laws of the state of Massachusetts. If you acquired the SOFTWARE outside of the United States, then local laws may apply.

Should you have any questions concerning this EULA, or if you desire to contact Quinn-Curtis, Inc. for any reason, please contact Quinn-Curtis, Inc. by mail at: Quinn-Curtis, Inc., 18 Hearthstone Dr., Medfield MA 02052 USA, or by telephone at: (508)359-6639, or by electronic mail at: support@Quinn-Curtis.com.

Table of Contents

Table of Contents vii
1. Introduction to QCMatPack 1
Tutorials 1
QCMatPack for Java Background 1
QCMatPack for Java Dependencies 2
Directory Structure 3
Chapter Summary 5
2. Class Architecture of QCMatPack 7
Major Design Considerations 7
QCMatPack for Java Class Summary 8
Matrix Viewer Classes 9
Complex number class 11
Matrix Classes 12
Solution of Linear Systems 13
Solution of Eigensystems 16
Multiple Regression 17
Curve Fitting 18
Digital Signal Processing 19
Class Hierarchy of QCMatPack 20
3. Complex Numbers 23
Complex Math Operators 24
Complex Math Functions 27
Miscellaneous Functions 31
4. Matrix Classes (DDMat and DXMat classes) 33
Linear Algebra Terminology 33
Matrix Constructors 35
Matrix Operators 37
Matrix Methods 41
A Summary of all DDMat, DXMat Methods 52
Chapter 5 - Solution of Linear Systems of Equations 59
Standard Interface for the Solution of Linear Equations 61
Gauss-Jordan 63
LU Decomposition 64
QR Factorization 65
Cholesky Decomposition 66
Chapter 6 – Eigenvalue and Eigenvector Solutions 69
Standard Interface for Eigenvalue and Eigenvectors Solutions 72
QL Algorithm for Real Symmetric Matrices 72
QR Algorithm for Real Unsymmetric Matrices 74
LR Algorithm for Complex Matrices 77
Chapter 7 – Multiple Regression 81
Least Squares Multiple Regression 82
Automatic Selection of Regression Variables 85
8. Curve Fitting 93
Polynomial Curve Fitting 93

Cubic Splines 100
9. Digital Signal Processing 105
FFT Basics 105
DSP Windowing 107
10. Matrix Viewers 117
11. Thermocouple Linearization 121
NIST ITS-90 121
Cold Junction Compensation 123
12. Error Handling 129
13. Using QCMatPack for Java to Create Applications and Applets 133
Eclipse 133
Java Applets 143
Selected Bibliography 149
INDEX 151

1. Introduction to QCMatPack

Tutorials

A simple tutorial that describe how to get started with the QCMatPack for Java charting software are found in Chapter 12 (*Using QCMatPack for Java to Create Applications and Applets*).

QCMatPack for Java Background

Numerical analysis is the art of calculating solutions to mathematical problems involving real and complex numbers. Man has been trying to measure and calculate his physical world since at the least the beginning of recorded history. Ancient Egypt and Summeria/Babylonia are examples of cultures that required sophisticated numerical methods routines in order to support the construction, religious and agricultural aspects of their societies.

The numerical algorithms in this software package all reside in the realm of continuous mathematics and use both real, and in many cases, complex numbers. Real numbers in this software are approximated using 8-byte *double* numeric objects, with complex numbers requiring two *double* objects. It is the goal, and art of numerical programming, to minimize the negative effects, and accumulated errors, resulting from the imperfect representation of infinite precision real numbers by finite precision floating point numbers.

Numerical algorithms involving statistics, the solution of equations, eigenvalues and digital signal processing were among the first developed for use on computers. Starting in the 1950's, these algorithms were primarily developed in FORTRAN (short for **For**mula **Tran**slating System*)* and starting in the 1960's, ALGOL (short for **ALGO**rithmic **L**anguage). The FORTRAN implementations seem to have won out, even though ALGOL's structured programming elements made it the superior language for algorithm expression. Most of the major, well known, numerical methods libraries, for execution on super, mainframe and even microprocessor-based computers are written in FORTRAN. These include EISPACK, LINPACK, LAPACK, BLAS, SLATEC and MINPACK. In many cases, algorithms were translated from ALGOL, into FORTRAN, often obscuring the logic and flow of the original algorithm. FORTRAN versions of the BLAS and LAPACK libraries are still used today in compiled form with modern computer languages. You are able to find C++ versions of these libraries that are nothing more than C++ wrappers around the original compiled FORTRAN code. You will even find these compiled libraries used with Java and the .Net languages, C# and VB, passing data through an interface that permits mixing unmanaged compiled libraries with managed, .Net code. The result, though, cannot be considered "managed" code.

This software implements what we consider the most useful numerical algorithms, written entirely in Java, with no calls to external libraries that do not utilize the Java

managed memory system. That includes routines for the generalized solution of linear algebra problems, and specific algorithms for solutions of linear equations, eigensystems, digital signal processing, multiple regression, and curve fitting. Since the algorithms are implemented using 100% managed code, they are the better choice for workstation and web based Java applications.

Many of the algorithms in science and engineering require the use of complex numbers in the form **a** + **b*i***, where **a** and **b** are real numbers and ***i*** represents the square root of -1. The Java runtime, at this time, does not include a numeric type, or object class that supports complex numbers and math; therefore one (**DComplex**) is included with this software. Building on the Java standard ***double*** numeric type, and our **DComplex** complex number class, two generalized matrix and linear algebra classes (**DDMat** and **DXMat**) were created. The numerical algorithms were then implemented using those matrix classes for input and output.

QCMatPack for Java Dependencies

The **com.quinncurtis.matpackjava** package is self-contained. It uses only standard classes that ship with the Java 1.2 API. In addition, all components referenced in the software are **lightweight** components. The software uses the major Java packages listed below.

Package java.awt

The **java.awt** package is the core of Java AWT (Abstract Windowing Toolkit). It contains all of the classes that create user interfaces and draw graphics and images. The **java.awt** classes most often used in the Quinn-Curtis **MatViewJava** classes are the **graphics**, **graphics2D**, **geom**, **color**, **font**, and **image**.

Class Java.awt.graphics

The **graphics** class is the abstract base class for all graphics. It contains the basic line drawing, text, area fill and color control functions needed to output a graph to an output device.

Class java.awt.color

Provides a class to define colors in terms of their individual RGB (Red, Green, Blue) components.

Class java.awt.font

This class provides routines for defining and manipulating character fonts.

Package java.util.*

This package provides classes for tree, list and vector management, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Package java.text.*

This package provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.

Package java.io.*

This package provides classes for file i/o and serialization.

Package javax.swing.event.*

This package provides for the **MouseInputListener** events fired by Swing components.

Package javax.swing

This package provides classes of "lightweight" (all-Java language) components that, in theory if not practice, work the same on all platforms.

.

Directory Structure

The QCMatPack for Java class library uses the following directory structure:

Drive:

Quinn-Curtis\ - Root directory

java\ - Quinn-Curtis Java based products directory

RedistributableFiles\ - contains the qcmatpackjava.jar file that you should use when redistributing applications that use this library. It does not contain the Javadoc documentation is much smaller than the version of the same file found in the \lib folder.

docs\ - Quinn-Curtis Java related documentation directory

lib\ - Quinn-Curtis Java related compiled libraries (jar) and components directory. The jar files in this folder contain Javadoc documentation for the classes in the jar file and are much larger than jar files of the same name in the RedistributableFiles folder.

com

quinn-curtis

matpackjava

Examples\ - Java examples directory

ComplexMath – A simple example demonstrating the use of the **DComplex** complex number class.

QCMatPackExample1 - The tutorial example program.

EigenRealSym\ - Eigenvalue/Eigenvector solutions for real symmetric matrices using the Householder QL and Hessen QR algorithms.

EigenRealUnsym\ - Eigenvalue/Eigenvector solutions for real unsymmetric matrices using the Hessen QR algorithm.

EigenComplex\ - Eigenvalue/Eigenvector solutions for complex matrices.using the Hessen LR algorithm.

SimpleLES\ - A very simple example demonstrating the solution of systems of equations using real numbers.

LESTest\ - Solving systems of equations (real numbers) using Cholesky, LU Decomposition, QR Factorization and Gauss-Jordan algorithms.

SimpleLESX\ - A very simple example demonstrating the solution of systems of equations using complex numbers.

XLESTest\ - Solving systems of equations (complex numbers) using Cholesky, LU Decomposition, QR Factorization and Gauss-Jordan algorithms.

MulRegTest\ - A multiple regression test program.

StepRegTest\ - A stepwise multiple regression test program.

CurveFitTest\ - Curve fitting solutions using polynomial curve fitting

CubicSplinesTest\ - Discrete function approximation using cubic splines.

FFTTest\ - Calculating FFTs and related functions

TCLinTest\ - A thermocouple linearization program.

MatErrorTest\ - Demonstration of QCMatPack error handling

There are two versions of the **QCMatPack** class library: the 30-day trial versions, and the developer version. Each version has different characteristics that are summarized below:

30-Day Trial Version

The trial version of **QCMatPack for Java** is downloaded as a zip file named Trial_QCMatPackR1x7x.zip. Once you download the zip file, you un-zip it to a local hard drive and run the Setup.exe program from the resulting QCMatPackInstall folder. The 30-day trial version stops working 30 days after you run the Setup.exe program for the first time.

Developer Version

The developer version of QCMatPack for Java is downloaded as a zip file, name something similar to JAVMATDEV1R1x0x353x1.zip. Once you download the zip file, you un-zip it to a local hard drive and run the Setup.exe program from the resulting QCMatPackInstall folder. You can use your original download links to download free updates for a period of 2-years.

Chapter Summary

The remaining chapters of this book discuss the **QCMatPack for Java** numerical analysis software designed to run on any hardware that has Java 1.4 or higher interpreter, available for it.

Chapter 2 presents the overall class architecture of the **QCMatPack for Java** and summarizes all of the classes found in the software.

Chapter 3 describes complex number class used throughout the software.

Chapter 4 describes the real and complex number matrix classes that you will use to get data in and out of the numerical analysis routines.

Chapter 5 describes the classes that solve systems of linear equations for real and complex numbers. Algorithms include Cholesky, LU Decomposition, QR Factorization and Gauss-Jordan.

Chapter 6 describes the general eigenvalue and eigenvector problem**.** Algorithms are presented for matrices that are real-symmetric (QL and QR), real-unsymmetric (QR), and complex. (LR).

Chapter 7 describes a least squares multiple regression class that handles multiple regression and stepwise multiple regression, along with related regression summary statistics.

Chapters 8 describes a curve-fitting class that handles polynomial, exponential, rational polynomial and mixed curve-fitting. It also describes a cubic splines class.

Chapters 9 describes a digital signal processing class (DSP) that solves for real and complex FFT's, power spectrums, and related parameters.

Chapters 10 describes the matrix viewer controls used to display real and complex matrices in a Windows form.

Chapters 11 describes a thermocouple linearization class based on the most current standards: NIST ITS-90.

Chapter 12 discussed errors that the software can generate and the various error processing modes.

Chapter 13 is a tutorial that describes how to use **QCMatPack** to create Java Applications and Applets.

2. Class Architecture of QCMatPack

Major Design Considerations

This chapter presents an overview of the **QCMatPack for Java** class architecture. It discusses the major design considerations of the architecture:

- **QCMatPack** is 100% managed code, using only the Java core numeric routines in the matrix and numerical analysis classes. The software makes NO calls to external, non-Java based libraries.

- Simple to use matrix and numerical analysis classes for the most common problems in science, engineering, industry, and business.

- There are no limits regarding the size of vectors and matrices, other than the amount of free memory on the host computer. There are no limits regarding the size of the vectors and matrices used in the solution of general linear systems, solution of eigenvalue problems, multiple regression, curve fitting and DSP (FFT) routines.

The chapter also summarizes the classes in the **QCMatPack for Java** library.

There are four primary features of the overall architecture of the **QCMatPack for Java** classes. These features address major shortcomings in existing numerical software for use with both Java and other computer languages.

- First, **QCMatPack** implements a complex number class, **DComplex**, that provides a data structure, functions and operators for manipulating complex numbers. Operator overloading of the +, -, *, /, (etc.) operators make manipulating complex numbers as simple as possible.

- Second, **QCMatPack** implements several generalized matrix classes, the most important of which are the **DDMat** for real number matrices and the **DXMat** class for complex number matrices. These are used for the input and output of data to the numerical analysis routines and the manipulation of real and complex number matrices. The matrix classes are designed so that there is no practical limit, other than the amount of free system memory, to the size of matrices. The **DDMat** and **DXMat** are symmetrical, in that the same matrix math and linear algebra routines are available for both real and complex numbers.

- Third, the library implements the best of the algorithms we could find in numerical analysis. These include LU Decomposition, QR Factorization, Cholesky and Gauss-Jordan for the solution of real and complex linear systems; QL, QR, and LR routines for the solution of real and complex eigenvalue/eigenvector problems; FFT's for real

and complex data, multiple; and stepwise multiple regression, curve fitting, and thermocouple linearization.

- Fourth, a pair of matrix viewer controls, one for real (**DMatViewer**) and the other for complex matrices (**XMatViewer**), let you add scrollable matrices to your Windows form-based application programs.

QCMatPack for Java Class Summary

The following categories of classes realize these design considerations.

Complex number class

The **DComplex** class implements a complex number class using a pair of doubles to hold the real and imaginary parts of a complex number. Extensive support for support of complex numeric functions and operator overloading, including conjugate, +, -, +, /, magnitude, power, exponent, and square root functions.

Matrix classes

Two matrix classes, **DDMat** for real numbers, and **DXMat** for complex numbers, simplify data manipulation and the input/output of data to/from the numerical analysis routines.

Solution of linear systems

There is a symmetrical group of eight classes for the solution of real and complex linear systems. These classes correspond to real and complex versions of Cholesky factorization (**LESCholesky**, **LESXCholesky**), LU Decomposition (**LESLU**, **LESXLU**), QR Factorization (**LESQR**, **LESXQR**) and Gauss-Jordan (**LESGJ**, **LESXGJ**). Each class provides for the solution of one right hand side, multiple right hand sides and matrix inverse.

Solution of eigensystems

The eigensystem problem is divided into three categories real-symmetric, real-unsymmetric and complex (both symmetric (Hermitian) and complex unsymmetric. The real-symmetric case is handled using the QL class **EigenQL**. The **EigenQR and EigenQR2** classes solve the real-unsymmetric case. And the **EigenXLR** class will solve general complex eigenvalue problems.

Multiple regression

General multiple regression, forward regression, backward regression, stepwise multiple and split regression and regression summary statistics are found in the **MulReg, StepwiseMulReg and SplitMulReg** classes.

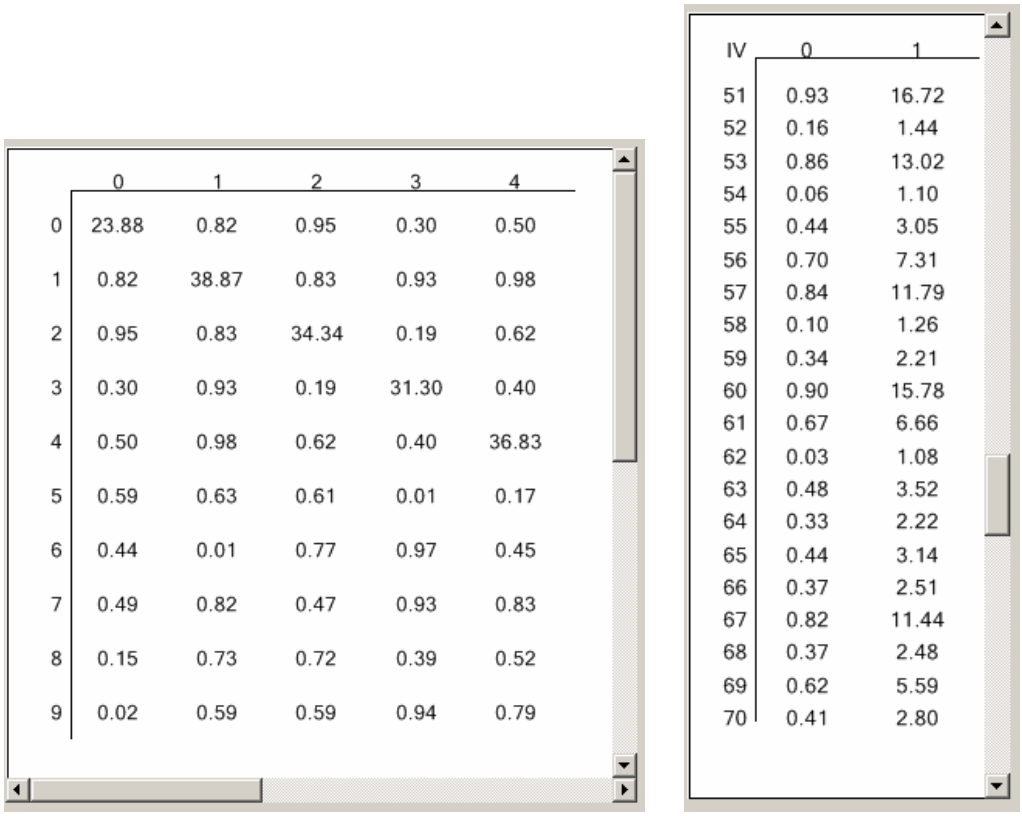
Curve fitting	The **CurveFit** class includes support for polynomial, exponential, rational polynomial, and mixed curve fitting. The **CubicSplines** class uses a cubic splines algorithm for discrete function evaluation.
Digital signal processing	The **DSP** class includes routines for real and complex FFT's and power spectrum calculations. Auxiliary routines calculate FFT magnitudes, phases and frequencies.
Matrix viewer classes	The **DMatViewer** and **XMatViewer** classes are **UserControl** subclasses that display real and complex number matrices, respectively.
Thermocouple linearization	The thermocouple linearization routines will convert from millivolts to temperature and temperature to millivolts, for B, E, J, K, N, R, S, T thermocouple types. Calculations are is based on the NIST ITS-90 standard as published in the NIST Monograph 175.

A summary of each category appears in the following section.

Matrix Viewer Classes

System.Windows.Forms.UserControl
    MatView
        DMatViewer
        XMatViewer

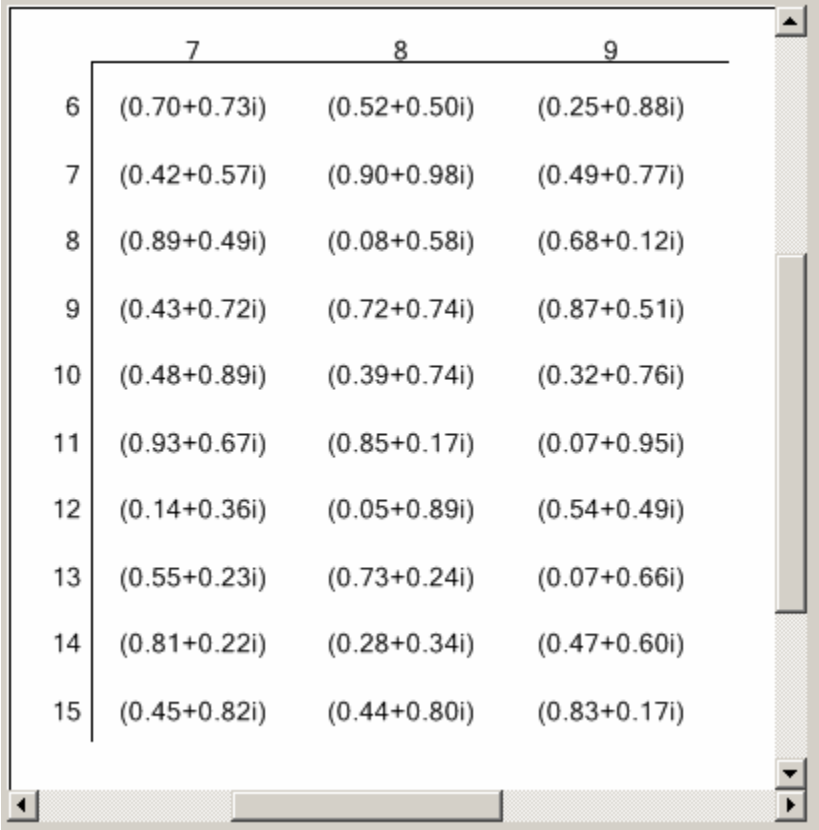
The matrix viewer classes display a matrix in a Java **JPanel** derived object. As such, it can be placed, and resized anywhere on a JPanel derived form. If the matrix is larger than the size you have created the view, scrollbars give the option of scrolling the rows and columns of the matrix. The **DMatViewer** class displays real matrix while the **XMatViewer** displays a complex matrix. The matrix viewers include options to edit matrix values, change font characteristics, the numeric format of the matrix element values, and the number of displayed rows and columns.



	0	1	2	3	4
0	23.88	0.82	0.95	0.30	0.50
1	0.82	38.87	0.83	0.93	0.98
2	0.95	0.83	34.34	0.19	0.62
3	0.30	0.93	0.19	31.30	0.40
4	0.50	0.98	0.62	0.40	36.83
5	0.59	0.63	0.61	0.01	0.17
6	0.44	0.01	0.77	0.97	0.45
7	0.49	0.82	0.47	0.93	0.83
8	0.15	0.73	0.72	0.39	0.52
9	0.02	0.59	0.59	0.94	0.79

IV	0	1
51	0.93	16.72
52	0.16	1.44
53	0.86	13.02
54	0.06	1.10
55	0.44	3.05
56	0.70	7.31
57	0.84	11.79
58	0.10	1.26
59	0.34	2.21
60	0.90	15.78
61	0.67	6.66
62	0.03	1.08
63	0.48	3.52
64	0.33	2.22
65	0.44	3.14
66	0.37	2.51
67	0.82	11.44
68	0.37	2.48
69	0.62	5.59
70	0.41	2.80

DMatViewer displays matrix elements of a DDMat (real) matrix



	7	8	9
6	(0.70+0.73i)	(0.52+0.50i)	(0.25+0.88i)
7	(0.42+0.57i)	(0.90+0.98i)	(0.49+0.77i)
8	(0.89+0.49i)	(0.08+0.58i)	(0.68+0.12i)
9	(0.43+0.72i)	(0.72+0.74i)	(0.87+0.51i)
10	(0.48+0.89i)	(0.39+0.74i)	(0.32+0.76i)
11	(0.93+0.67i)	(0.85+0.17i)	(0.07+0.95i)
12	(0.14+0.36i)	(0.05+0.89i)	(0.54+0.49i)
13	(0.55+0.23i)	(0.73+0.24i)	(0.07+0.66i)
14	(0.81+0.22i)	(0.28+0.34i)	(0.47+0.60i)
15	(0.45+0.82i)	(0.44+0.80i)	(0.83+0.17i)

XMatViewer displays matrix elements of a DXMat (complex) matrix

Complex number class

DComplex

In mathematics, a **complex number** is a number of the form

$$a + bi$$

where *a* and *b* are real numbers, and *i* is the imaginary unit, with the property $i^2 = -1$. The real number *a* is called the *real part* of the complex number, and the real number *b* is the *imaginary part*. Real numbers may be considered to be complex numbers with an imaginary part of zero; that is, the real number *a* is equivalent to the complex number *a*+0*i*. (source Wikipedia).

This software represents a complex number using the **DComplex** class. It represents the real and imaginary parts of a complex number using two *double* values. It includes overrides of the most common math operators (+, -, *, /, =), making it very simple and logical to write mathematical expressions using complex numbers. For example, the following line in bold is a valid expression involving **DComplex** complex number arguments.

```
DComplex c1 = new DComplex(1.0, 2.0);

DComplex c2 = new DComplex(3.0, 4.0);

double r1 = 5.0;

//      c3 = c1 * c2 * r1 + 6.0;

DComplex c3 = DComplex.add(DComplex.multiply(DComplex.multiply(c1,c2), r1), 6.0);
```

The **DComplex** class also includes a large number of mathematical functions used in conjunction with complex numbers (Conjugate, Magnitude, Abs, Sqrt, Exp, Pow, and Arg).

The **DXMat** class is the main class used to represent vectors and matrices of **DComplex** complex numbers. The linear equations classes: **LESXLU**, **LESXGJ**, **LESXCholesky** and **LESXQR,** are specific to the **DComplex** complex number type. The eigensystem classes: **EigenQR2**, **EigenXLR,** also uses the **DComplex** number type.

Matrix Classes

MatBase

DMatBase

DDMat

XMatBase

DXMat

The **MatBase** abstract base class is the base class for all matrix types. The **DMatBase** class is the abstract representation of a real matrix. The **DDMat** class is the concrete representation of a real matrix, using doubles for numeric storage. The **XMatBase** class is the abstract representation of a complex number matrix. The **DXMat** class is the concrete representation of a complex number matrix, using **DComplex** objects for numeric storage.

MatBase The **MatBase** abstract base class is the base class for all matrix types. It defines the dimensions of a matrix, usually

referred to as m x n, representing the number of rows and columns of a matrix. It also contains numeric constants commonly used in matrix math routines.

DMatBase — The **DMatBase** class is the abstract representation of a real matrix. It contains a large number of methods specific to real matrices. At this time, the only matrix class that derives from **DMatBase** is the concrete matrix class **DDMat**. Future versions of the software may define one or more real sparse matrix classes derived from **DMatBase**.

XMatBase — The **XMatBase** class is the abstract representation of a complex matrix. It contains a large number of methods specific to complex matrices. At this time, the only matrix class that derives from **XMatBase** is the concrete matrix class **DXMat**. Future versions of the software may define one or more complex sparse matrix classes derived from **XMatBase**.

DDMat — The **DDMat** class is the concrete representation of a real matrix, using doubles for numeric storage. It contains a large number of matrix manipulation methods optimized for dense double storage. It also includes matrix math methods for the (+, -, *, /, etc.) operators.

DXMat — The **DXMat** class is the concrete representation of a complex matrix, using **DComplex** objects for numeric storage. It contains a large number of matrix manipulation methods optimized for dense **DComplex** object storage. It also includes matrix math methods for the (+, -, *, /, etc.) operators.

Solution of Linear Systems

LESBase
- **LESLU**
- **LESGJ**
- **LESCholesky**
- **LESQR**

LESXBase
- **LESXLU**
- **LESXGJ**
- **LESXCholesky**
- **LESXQR**

A system of linear equations is defined by the linear algebra equation:

$$Ax = b$$

where A, x and b represent real/complex matrices. A is the known coefficient matrix, b is the known right hand side, and x is the unknown solution. A is usually square, though in the case of the QR algorithm A can be over determined where the number of rows is greater than the number of columns. The matrices x and b are usually column matrices of one column, though in the case of multiple right hand sides, both x and b can have multiple columns.

This software provides a symmetrical group of classes for the solution of real and complex linear systems.

LESBase — This class is the abstract base class for linear equation solvers that use real numbers. It defines a standard interface for solving systems of equations that involve one or more right hand sides.

LESXBase — This class is the abstract base class for linear equation solvers that use complex numbers. It defines a standard interface for solving systems of equations that involve one or more right hand sides.

There are eight concrete classes, derived from **LESBase** and **LESXBase**, corresponding to real and complex versions of Cholesky factorization (**LESCholesky**, **LESXCholesky**), LU Decomposition (**LESLU**, **LESXLU**), QR Factorization (**LESQR**, **LESXQR**) and Gauss-Jordan (**LESGJ**, **LESXGJ**). Each class provides for the solution of one right hand side, multiple right hand sides and the matrix inverse.

LESLU — The **LESLU** class uses the LU decomposition algorithm to solve a system of real number linear equations**.** It is a very stable, efficient, method of solving systems of equations. It is particularly useful for solving systems of equations involving multiple right hand sides, since the LU decomposition of the primary coefficient matrix only needs to be solved for once. From that point on, multiple right hand sides can be solved for using the decomposed matrix and simple back-substitution.

LESXLU — The **LESXLU** class is the complex number version of the **LESLU** class, using the LU decomposition algorithm to solve a system of complex number linear equations**.**

LESGJ

The **LESGJ** class uses the Gauss-Jordan with partial pivoting algorithm to solve a system of real number linear equations**.** It is a very stable, efficient, method of solving systems of equations.

LESXGJ

The **LESXGJ** class is the complex number version of the **LESGJ** class, using the Gauss-Jordan with partial pivoting algorithm to solve a system of complex number linear equations**.**

LESCholesky

The **LESCholesky** class uses the Cholesky decomposition algorithm to solve a system of real number linear equations**.** The Cholesky algorithm requires that the coefficient matrix is a symmetrical (where the elements below the diagonal are the transpose of the elements above the diagonal), positive definite, matrix. It turns out that this class of matrices are quite common in systems of equations, so this algorithm should be given special consideration. It is up to twice as fast as LU decomposition and four times as fast as Gauss-Jordan. Like LU decomposition, it is particularly useful for solving systems of equations involving multiple right hand sides, since the Cholesky decomposition of the primary coefficient matrix only needs to be solved for once. From that point on, multiple right hand sides can be solved for using the decomposed matrix and simple back-substitution.

LESXCholesky

The **LESXCholesky** class is the complex number version of the **LESCholesky** class, using the Cholesky decomposition algorithm to solve a system of complex number linear equations**.** The complex version of the Cholesky algorithm requires that the source coefficient matrix is Hermitian (the complex version of a symmetric matrix, where the elements below the diagonal are the complex conjugate of the elements above the diagonal, and the diagonal is real), positive and definite.

LESQR

The **LESQR** class uses the QR factorization algorithm to solve a system of real number linear equations**.** It is a very stable, efficient, method of solving systems of equations. It has one major advantage over the other equation solvers, in that it works with both square, and an over-determined (where the number of rows is greater than the number of columns, m > n) coefficient matrices. Like the LU and Cholesky algorithms, it is efficient at solving multiple right hand sides, since the QR factorization of the primary coefficient matrix only needs to be solved for once. From

that point on, multiple right hand sides can be solved for using the factored matrix and simple back-substitution.

LESXQR — The **LESXQR** class is the complex number version of the **LESQR** class, using the QR factorization algorithm to solve a system of complex number linear equations**.**

Solution of Eigensystems

EigenBase
- **EigenQL**
- **EigenQR**
- **EigenQR2**

EigenXBase
- **EigenXLR**

The classes above implement algorithms for the solution of real and complex eigensystems. The **EigenBase** class is the abstract representation of an eigensystem solver that starts with a coefficient matrix of real numbers. The **EigenXBase** class is the abstract representation of an eigensystem solver that starts with a coefficient matrix of complex numbers.

EigenBase — This class is the abstract base class for eigensystem solvers that start with real coefficient matrix. It defines a standard interface for solving for eigenvalues and eigenvectors for real, symmetric and unsymmetric matrices.

EigenXBase — This class is the abstract base class for eigensystem solvers that start with complex coefficient matrix. It defines a standard interface for solving for eigenvalues and eigenvectors for complex, symmetric (Hermitian) and unsymmetric (un-Hermitian) matrices.

.

There are four concrete classes, derived from **EigenBase** and **EigenXBase**, that solve for all general categories of the initial eigensystem matrix: real-symmetric, real-unsymmetric, complex-symmetric (Hermitian actually) and complex unsymmetric (non-Hermitian.).

EigenQL — Use this class to solve for the eigenvalues and eigenvectors of real, symmetric, matrices. It uses the implicit QL algorithm and can solve real-symmetric eigensystem problems, which generate real-only eigenvalue and eigenvector solutions.

EigenQR	Use this class to solve for the real and/or complex eigenvalues of general (not necessarily symmetric), real, matrices. The QR algorithm can solve general eigensystem problems, that generate both real and complex eigenvalue and eigenvector solutions.
EigenQR2	Use this class to solve for the real and/or complex eigenvalues and eigenvectors of general real matrices. It uses the QR algorithm and can solve general (symmetric and non-symmetric) eigensystem problems, which generate complex eigenvalue and eigenvector solutions.
EigenXLR	Use this class to solve for the eigenvalues and eigenvectors of general complex matrices. It uses the LR algorithm and can solve general (Hermitian and non-Hermitian) eigensystem problems, which generate complex eigenvalue and eigenvector solutions.

Multiple Regression

MulReg
- **StepwiseMulReg**
- **SplitMulReg**

Regression analysis is a statistical tool for the evaluation of the relationship of one or more independent variables X_1, X_2,...,X_k to a single dependent variable Y for a fixed number of observations. The primary goal of regression analysis is to obtain predictions of Y using the known values of X. These predictions are calculated using a regression equation. They are subject to error and are only estimates of the true values of Y. There can be other goals of regression analysis, for example, to determine which of several independent variables are important and which are not for describing and predicting a dependent variable.

The **MulReg** class solves the generalized least-squares multiple regression problem. The **StepwiseMulReg** class includes forward, backward, and stepwise multiple regression algorithms that only includes independent variables (the x-values) in the final solution that pass a F-statistic significance test. The **SplitMulReg** splits the original data into two, develops a model using one dataset and uses that to predict the values in the second dataset.

MulReg The least-squares multiple regression algorithm calculates the coefficients for a set of specific independent variable (x-values). The method also calculates a large number of parameters describing the quality of the regression: the R value, R-squared value, standard error of the estimate (SEE), the overall F-Stat value, and the standard error and the F-Stat value for each regression equation coefficient.

StepwiseMulReg The stepwise multiple regression algorithm includes forward, backward and stepwise regression algorithms for automatic selection of regression variables. Stepwise regression removes variables that fail an F-statistic significance test, usually leaving you with fewer variables, and a simpler model, than you started with.

SplitMulReg The **SplitMulReg** splits the original data into two, develops a model using one dataset and uses that to predict the values in the second dataset.

Curve Fitting

CurveFit
CubicSplines

Experimental data often takes the form of measurements of some kind (temperature or volts for example) taken at discrete and possibly periodic times. In such cases, a table of the measured variable versus time is produced. The goal of the research is to uncover the underlying function f(x) which relates temperature to time. Once the underlying function is discovered (the solution is usually only an approximation of the true function), the function can be used to calculate the values of f(x) (temperature in this case) for x values (points in time) which were not part of the original measured data.

The **CurveFit** algorithms create a single model that is the least squares fit for all the data points in the original dataset. The **CubicSplines** class creates a family of cubic equations, one cubic equation for each sample interval, that can be used to interpolate y-values based on x-values not in the original dataset.

CurveFit The **CurveFit** algorithm calculates the coefficients for a single equation that fits all of the data points in the original

dataset. It supports four different algorithms for curve fitting: polynomial, exponential, rational fraction, and mixed (a combination of the first three methods).

CubicSplines — The **CubicSplines** class creates a family of cubic equations, one cubic equation for each sample interval, that can be used to interpolate y-values based on x-values not in the original dataset.

Digital Signal Processing

DSP

A **fast Fourier transform** (**FFT**) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFTs are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.

The DSP class is built around the Cooley-Tukey radix-2 FFT algorithm. Support routines (windowing, magnitude, frequency and phase calculations) and popular variants (power spectrum calculations, real-only FFTs) are also included.

DSP — The **DSP** class includes methods for **DDMat** (real-only) and **DXMat** (complex) FFT calculations. It also includes static FFT functions that accept double [] array arguments for maximum speed. The raw signal can be processed with a DPS window (NOWIN, RECTANGULAR, GAUSS, HAMMING, HANN, BARTLETT, BARTLETT_HANN, NUTTALL, BLACKMAN_HARRIS, BLACKMAN_NUTTALL, FLATTOP, PARZEN, WELCH, BLACKMAN, KAISER_BESSEL TRIANGULAR, and EXPONENTIAL.) before the FFT, and the FFT results can be interpreted using magnitude, phase, and frequency routines. The **PowerSpectrum** method returns the power spectrum of the original signal.

Miscellaneous Chart Classes

BMTimer
DTriplet
XTriplet

MatError
MatSupport
RegStats
IVect

Various classes exist to support the main matrix and algorithm classes

BMTimer	A simple timer class used to benchmark algorithms in the software. It is based on the system clock and while it has microsecond resolution, its accuracy is only 20 milliseconds.
DTriplet	The **DTriplet** class is used to transfer data to/from real matrices. It specifies a row, column and value.
XTriplet	This **XTriplet** class is the **DComplex** version of **DTriplet**
MatError	Processes errors in the matrix and algorithm classes.
MatSupport	Common functions used by multiple classes are moved into the **MatSupport** class, avoiding duplication.
RegStats	The **RegStats** class is used to return summary statistics for the multiple regression and curve fitting classes.
IVect	IVect is a simple integer vector class used to pass integer data to and from some classes.

Class Hierarchy of QCMatPack

DComplex
MatBase
    DMatBase
        DDMat
    XMatBase
        DXMat
MatViewerBase
    DMatViewer
    XMatViewer
LESBase
    LESLU

- **LESGJ**
- **LESCholesky**
- **LESQR**

LESXBase

- **LESXLU**
- **LESXGJ**
- **LESXCholesky**
- **LESXQR**

EigenBase

- **EigenQR**
- **EigenQL**
- **EigenQR2**

EigenXBase

- **EigenXLR**

MulReg

- **StepwiseMulReg**
- **SplitMulReg**

CurveFit
CubicSplines
DSP
TCLin
BMTimer
DTriplet
XTriplet
MatError
MatSupport
RegStats
IVect

3. Complex Numbers

The linear algebra, solution of linear systems and eigenvalue/eigenvector algorithms in this software come in versions that apply to both real and complex numbers. While the . *double* numeric type is what we use for real numbers, Java does not have a standard equivalent numeric type for complex numbers. To that end we have created our own complex class, **DComplex**, for use in our algorithms and your programs.

DComplex

In mathematics, a **complex number** is a number of the form

$$a + bi$$

where a and b are real numbers, and i is the imaginary unit, with the property $i^2 = -1$. The real number a is called the *real part* of the complex number, and the real number b is the *imaginary part*. Real numbers may be considered to be complex numbers with an imaginary part of zero; that is, the real number a is equivalent to the complex number $a+0i$. (source Wikipedia: http://en.wikipedia.org/wiki/Complex_number)

The **DComplex** class represents the real and imaginary parts of a complex number using two *double* values. It includes overrides of the most common math operators (+, -, *, /, =), making it very simple and logical to write mathematical expressions using complex numbers. For example, the following line in bold is a valid expression involving **DComplex** complex number arguments.

```
DComplex c1 = new DComplex(1.0, 2.0);

DComplex c2 = new DComplex(3.0, 4.0);

double r1 = 5.0;

//      c3 = c1 * c2 * r1 + 6.0;

DComplex c3 = DComplex.add(DComplex.multiply(DComplex.multiply(c1,c2), r1), 6.0);
```

The **DComplex** class also includes a large number of mathematical functions used in conjunction with complex numbers (Conjugate, Magnitude, Abs, Sqrt, Exp, Pow, and Arg).

The **DXMat** class is the main class used to represent vectors and matrices of **DComplex** complex numbers. The linear equations classes: **LESXLU**, **LESXGJ**, **LESXCholesky** and **LESXQR,** are specific to the **DComplex** complex number type. The eigensystem classes: **EigenQR** and **EigenQR2**, **EigenXLR**, also uses the **DComplex** number type.

DComplex constructors

Create **a DComplex** object using a pair of doubles, a single double, or another **DComplex** object.

DComplex(DComplex c) Constructor for a complex number.
DComplex(double re) Constructor for a complex number.
DComplex(double re, double im) Constructor for a complex number.

re The real part of the complex number.

im The imaginary part of the complex number.

You can also specify a complex number using the polar coordinates convention using the **FromPolar** static method.

```
public static DComplex fromPolar(double mag,
                                 double angle)
```

Access the real and imaginary part of the complex number using the member variables r and i, or the setRe, getRe, setIm, getIm methods.

```
DComplex c = new DComplex(1.0, 2.0);
c.r = 5.0;
c.i = 7.0;
double r2 = c.getRe();
double i2 = c.getIm();
```

Complex Math Operators

Complex numbers are added, subtracted, multiplied, and divided in a manner analagous to that of real numbers, following the associative, commutative and distributive laws of algebra, together with the equation $i^2 = -1$. Since the standard Java arithmetic operators (+, -, *, /) do not automatically work with user defined classes, the **DComplex** class includes these functions so that you can use a mix of **DComplex** and double values in math expressions.

Special note: Java does not have the ability to process overloaded operators. As a result, complex math operators are implemented as Java static method calls.

static DComplex	**add**(DComplex a, DComplex b)
static DComplex	**add**(DComplex a, double d)
static DComplex	**add**(double d, DComplex a)

$$(a+bi)+(c+di)=(a+c)+(b+d)i$$

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(3.0, 4.0);
double r1 = 5.0;

//  c3 = c1 + c2 + r1 + 6.0;
DComplex c3 = DComplex.add(DComplex.add(c1,c2), r1 + 6.0);
```

static DComplex	**subtract**(DComplex a, DComplex b)
static DComplex	**subtract**(DComplex a, double d)
static DComplex	**subtract**(double d, DComplex a)

$$(a+bi)-(c+di)=(a-c)+(b-d)i$$

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(3.0, 4.0);
double r1 = 5.0;
//  c3 = c1 - c2 + r1 - 6.0;
DComplex c3 = DComplex.add(DComplex.subtract(c1,c2), r1 - 6.0);
```

static DComplex	**multiply**(DComplex a, DComplex b)
static DComplex	**multiply**(DComplex a, double d)
static DComplex	**multiply**(double d, DComplex a)

$$(a+bi)(c+di)=ac+bci+adi+bdi^2=(ac-bd)+(bc+ad)i$$

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(3.0, 4.0);
double r1 = 5.0;
//  c3 = c1 * c2 * r1 + 6.0;
DComplex c3 = DComplex.add(DComplex.multiply(DComplex.multiply(c1,c2), r1), 6.0);0
```

static DComplex	**divide**(DComplex a, DComplex b)
static DComplex	**divide**(DComplex a, double d)
static DComplex	**divide**(double a, DComplex d)

$$(a + bi)/(c + di) = \left(\frac{ac + bd}{c^2 + d^2}\right) + \left(\frac{bc - ad}{c^2 + d^2}\right) i$$

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(3.0, 4.0);
double r1 = 5.0;
// c3 = (c1 / c2) / r1 + 6.0;
DComplex c3 = DComplex.add(DComplex.divide(DComplex.divide(c1,c2), r1), 6.0);
```

Mix all of the operators in complex math expressions

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(3.0, 4.0);
DComplex c3 = new DComplex(5.0, 6.0);
DComplex c4 = new DComplex(7.0, 8.0);
double r1 = 9.0;
// c5 = ((((c1 / c2)* c3) + 11.0)/ c4) - r1;
DComplex c5 =
DComplex.subtract(DComplex.divide(DComplex.add(DComplex.multiply(DComplex.divide(c
1,c2), c3), 11.0), c4), r1);
```

.

Equality Operators

Two complex numbers are equal if the real parts of the two numbers are equal, and the imaginary parts of the two numbers are equal.

static boolean	**isEqual**(DComplex a, DComplex b)

	Determine whether two complex numbers are equal.
`static boolean`	**`isEqual`**`(DComplex a, DComplex b, double tolerance)` Determine whether two complex numbers are almost (i.e. within tolerance.

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(1.0, 2.0);
if ( DComplex.isEqual(c1,c2)) // not true based on the values above
{
    label7.setText( "== Comparison ->  " +
            c1.toString() +
            " is equal to " +
            c2.toString());
}
```

`static boolean`	**`notEquals`**`(DComplex a, DComplex b)`

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = new DComplex(3.0, 4.0);
if (DComplex.notEquals(c1, c2))  // true based on the values above
{
    label8.setText( "!= Comparison ->  " +
        c1.toString() +
        " is not equal to " +
        c2.toString());
}
```

The other equality operators <, <=, >, and >= are not applicable to complex numbers. That is because you cannot compare two complex numbers and decide if one is greater than the other. For example, which is the larger number (100 + 5i), (5 + 100i), (5 – 100i), (100 – 5i). At best you can only compare the magnitudes of complex numbers. In that case, the complex numbers in the previous example are all equal, but fail the equality test ==. If you find you need to do some sort of greater than or less than test on two complex numbers, it is up to you to decide whether you look at: the real parts, the complex parts, or the magnitude of the complex number.

Complex Math Functions

The **DComplex** class also include many functions that need to be explicity calculated for complex numbers. These include the complex conjugate, magnitude (same as absolute value and modulus), normalization, polar angle, square root, raise to a power, and

exponent.(e to a complex number). Each function comes in two forms: the first is a static function that operates on an argument that is an instance of a **DComplex** class. The second is a **DComplex** member function that operates on its own instance variables.

DComplex.conj Method

The **complex conjugate** of the complex number $z = a + bi$ is defined to be $a - bi$

Get the conjugate of a complex number

DComplex	**conj**()
static DComplex	**conj**(DComplex a)

```
DComplex c1 = new DComplex(1.0, 2.0);
DComplex c2 = c1.conj();
DComplex c3 = DComplex.conj(c1);
```

The complex conjugate of c1 is (1.0 – 2.0i).

DComplex.abs Method

The **absolute value** (or *modulus* or *magnitude*) z of a complex number (a + bi) is $|z| = \sqrt{a^2 + b^2}$.

Calculate the absolute value of a complex number

double	**abs**()
static double	**abs**(DComplex c)

```
DComplex c1 = new DComplex(3.0, 4.0);
double r1 = c1.abs();
double r2 = DComplex.abs(c1);
```

The absolute value of c1 is a real number: DComplex.Abs(3 + 4i) = 5.

DComplex.normalize Method

The normalize method scales the complex number so that it has a magnitude of 1. The normalized value z of the complex number (a + bi) is ((a/Magnitude(a + bi) + b/Magnitude(a + bi) i.

DComplex	**normalize**()
static DComplex	**normalize**(DComplex a)

```
DComplex c1 = new DComplex(3.0, 4.0);
DComplex c2 = c1.normalize();
DComplex c3 = DComplex.normalize(c1);
```

The normalized value of c1 is: (3.0/5 + 4.0/5 i) = (0.6 + 0.8i)

DComplex.getPolarAngle Method

The **polar angle** r , in radians, of a complex number (a + bi) is:

r = Math.Atan2(b,a)

Calculate the polar angle of a complex number.

double	**getPolarAngle**()
static double	**getPolarAngle**(DComplex a)

```
DComplex c1 = new DComplex(3.0, 4.0);
double r1 = c1.getPolarAngle();
double r2 = DComplex.getPolarAngle (c1);
```

The polar angle of c1 is: Math.Atan2(4.0, 3.0) = .9273 radians

DComplex.sqrt Method

Calculate the square root of a complex number.

DComplex	**sqrt**()
static DComplex	**sqrt**(DComplex c)
static DComplex	**sqrt**(double xr, double xi)

```
DComplex c1 = new DComplex(3.0, 4.0);
DComplex c2 = c1.sqrt ();
DComplex c3 = DComplex.sqrt (c1);
```

The square root of c1 is: (1.0 + 2.0i)

DComplex.pow Method

Raise a complex number to real power.

static DComplex	**pow**(DComplex c, double exponent)
DComplex	**pow**(double exponent)

```
double r1 = 2.0;
DComplex c1 = new DComplex(3.0, 4.0);
DComplex c2 = c1.pow(r1);
DComplex c3 = DComplex.pow (c1, r1);
```

The value of $c1^2$ is: (-7.0 + 24.0i)

DComplex.exp Method

Raise the value e to a complex power.

DComplex	**exp**()
static DComplex	**exp**(DComplex c)

```
double r1 = 2.0;
DComplex c1 = new DComplex(3.0, 4.0);
DComplex c2 = c1.pow(r1);
DComplex c3 = DComplex.pow (c1, r1);
```

The value of e^{c1} is: (-13.128783081462162 - 15.200784463067956 i)

Miscellaneous Functions

The **DComplex** class also some support functions which make the input and output of complex numbers easier: toString and parse.

DComplex.toString Method

This method converts the current value of the member variables into a string of the form "a + bi". The complex number (1.1 + 2.2i) is represented in string notation as "(1.1 + 2.2i)". The default numeric format of the two double values in the string are controlled by the "G" numeric format option of the double.toString(string) format method. A second toString method is supplied that can take the floating point format string as an argument.

Get the string representation of a complex number.

`java.lang.String`	**`toString`**`()`
`java.lang.String`	**`toString`**`(java.lang.String s)`

DComplex.Parse Method

This method converts a string in the format "a + bi" into a **DComplex** object.

Get the string representation of a complex number.

`static DComplex`	**`parse`**`(java.lang.String s)` Parse a complex representation in this fashion: "(%f, %f)"

4. Matrix Classes (DDMat and DXMat classes)

MatBase
- **DMatBase**
  - **DDMat**
- **XMatBase**
  - **DXMat**

There are two matrix classes you will use for the input and output of numeric data to the linear algebra, linear systems, eigenvalue/eigenvector, and FFT algorithms: **DDMat** for real matrices and **DXMat** for complex matrices. Using the matrix classes you can read and write to individual elements; add, delete, swap, copy, scale, read/write rows and columns of a matrix. The matrix classes also carry out fundamental linear algebra subroutines such as vector and matrix norms, dot products and saxpy operations. Matrices can be initialized an element at a time, from standard Java one-and two-dimensional double arrays (double [] and double [][]), from other matrices, and from files. The matrix classes also include matrix math methods (+, -, *, /) that permit matrix math. The matrix classes also double as vector classes, a matrix with a column or row dimension of one is considered a column or row vector respectively. In that case matrix elements can be accesses using 1-D vector brackets [], instead of the normal 2D matrix brackets [,].

The **MatBase**, **DMatBase** and **XMatBase** classes are the base classes of the **DDMat** and **DXMat** classes. See the hierarchy at beginning of the chapter. Since these classes are abstract they can not be instantiated. They do contain important static and instance properties and methods used in the derived **DDMat** and **DXMat** classes. The only matrix classes you need to know how to instantiate are these two classes.

The **DDMat** and **DXMat** classes are very symmetrical. The methods and properties of the two classes all have the same names to make converting from one to the other as simple as possible. The major difference between the two classes is that the **DDMat** generally uses *double* arguments and the **DXMat** generally uses complex (class **DComplex**) arguments. This rule is not absolute, since some complex matrix functions do use double arguments and return double values. Because of the symmetry between the two matrix classes, constructors and methods will be documented in parallel to save space.

Linear Algebra Terminology

Matrices and linear algebra use mathematical terms not in common use with the average programmer. In order to qualify the special characteristics of different types of matrices, we use many of these terms.

Matrix order – The order of a matrix is the dimensions of the matrix, i.e. the number of rows and columns it has. Two matrices are of the same order if they have the same dimensions. The number of rows in a matrix is usually referred to using the letter *m*, the number of columns the letter *n*.

Transpose – The transpose of a real matrix is formed by taking an (m x n) matrix and converting it to an (n x m) matrix, making the rows of the original matrix the columns of the transposed matrix. For example, the row elements of the original matrix [0][0], [0][1], [0][2], …[0][n-1] become the column elements of the transposed matrix [0][0], [1][0], [2][0], …[n-1][0].

While it is possible to transpose a complex matrix using the simple description above, it is not mathematically correct. The proper mathematical transpose of a complex matrix is more properly called the conjugate transpose. In a conjugate transpose, the complex conjugate of the source row values become the new column values of the destination matrix. For example, if element [5,3] of the source matrix has the complex value (7 + 2i), the element [3,5] of the transposed matrix will have the value (7 – 2i), the complex conjugate of the original value.

Symmetric Matrix – A real matrix is considered to be symmetrical if it is square, and the elements above the diagonal are the mirror image of the elements below the diagonal, i.e. A[i][j] = A[j][i]. For example, in a symmetric matrix, the following element pairs, one above the diagonal and the other below the diagonal, must be equal: [5][3] = [3][5], [4][1] = [1][4], [11][7] = [7][11],

While a complex matrix can be symmetric, A[i][j] = A[j][i], it is not a useful condition. Like the complex transpose, a mathematically symmetric complex matrix uses the complex conjugate. A truly symmetric complex matrix, known as a **Hermitian matrix**, requires that A[i][j] = Conjugate(A[j][i]). Using this definition, the diagonal values of a Hermitian matrix must be real (no imaginary component), since that is the only way a diagonal element can equal its own complex conjugate.

Positive Definite Matrix – a positive definite matrix is a **Symmetric (or Hermitian** for complex numbers**)** matrix that has the added property that all of its eigenvalues are positive. While it seems of little importance for the non-mathematician, the positive definite matrix shows up automatically in many linear algebra applications, including the solution of least squares problems (multiple regression) and the solution of coupled differential equations. Algorithms, Cholesky factorization in particular, have been developed explicitly to exploit the structure of positive definite matrices and these algorithms can be 2-4 times as fast as algorithms that work on more generalized matrices.

Orthogonal Matrix - an orthogonal matrix is a square matrix *Q* whose transpose is its inverse: $Q^T Q = QQ^T = I.$

Base-0 vs. Base-1 Matrices – Originally, most computer matrix math and linear algebra algorithms were written in FORTRAN and ALGOL. Those computer languages define the index range of their built in array types, dimension [m x n], as extending from [1..m][

1..n]. Matrices of this type use base-1 indexing. In contrast, C, C++, C#, and Java all use base-0 indexing, where the index range of an array with the dimensions [m x n] is [0..m-1][0..n-1]. Algorithms written for base-1 indexing will not work, and will generally crash programs written using compilers that require base-0 indexing.

This software assumes base-0 indexing. All matrices dimensioned to the size (m x n) use array indexing in the range [0..m-1][0..n-1]. The software includes routines that will convert a base-0 matrix into a base-1 matrix. In that case, the base-0 matrix dimensioned to size (m x n), with array indexing of [0..m-1][0..n-1], is converted to a matrix size (m + 1) x (n + 1) that can be used with 1-based indexing in the range [1..m][1..n]. All of the data in the original matrix is shifted one column and one row so that all data fits in the new index range of [1..m][1..n]. While the 0 row and column indices in these converted matrices are still useable, in a true base-1 algorithm, they would never be accessed, because they are considered out of range. A base-1 matrix can be converted back to a base-0 matrix. In that case the base-1 matrix, dimension (m + 1) x (n + 1) is converted back to a matrix of with the dimensions (m x n). The data is shifted back one row and one column, so that the data starts in the 0^{th} row and column.

Matrix Constructors

Define a matrix with the dimensions (*rows* x *columns*).

```
DDMat
DDMat(int rows, int columns)

DXMat
DXMat(int rows, int columns)
```

Construct a matrix as a copy of the *source* matrix.

```
DDMat
DDMat(DMatBase source)

DXMat
DXMat(XMatBase source)
```

Construct vector as the column or row (*rcflag* = V_ROW or VCOL) of the source matrix. The index of the column or row to copy is *rcnum*.

```
DDMat
```

```
DDMat(DDMat source, int rcflag, int rcnum)

DXMat
DXMat(DXMat source, int rcflag, int rcnum)
```

Construct a matrix with the dimension (*rows* x *columns)* and initialize it with triplets from the *trips* array.

```
DDMat
DDMat(DTriplet[] trips, int rows, int columns)

DXMat
DXMat(XTriplet[] trips, int rows, int columns)
```

Construct a vector initialized using the 1D *source* array. The **DXMat** matrix has a constructor the initializes the real and imaginary parts of a complex matrix using the *resource* and *imsource* Java 1D arrays.

```
DDMat
DDMat(double[] source)

DXMat
DXMat(double[] resource)
DXMat(double[] resource, double[] imsource)
DXMat(DComplex[] source)
```

Construct a matrix using the *source* Java 2D array. The **DXMat** matrix has a constructor the initializes the real and imaginary parts of a complex matrix using the *resource* and *imsource* Java 2D arrays.

```
DDMat
DDMat(double[][] source)

DXMat
DXMat(DComplex[][] resource)
DXMat(double [][] resource, double [][] imsource)
DXMat(double [][] resource)
```

Construct a matrix as a vector with the dimension *elements*.

```
DDMat
DDMat(int elements)

DXMat
DXMat(int elements)
```

Construct a matrix as a vector, dimension *elements*, and initialize all elements to the value *r*.

```
DDMat
DDMat(int elements, double r)

DXMat
DXMat(int elements, DComplex r)
```

Matrix Operators

Matrices are added, subtracted, multiplied, and divided following the rules of linear algebra.

Matrix Addition

static DDMat	**add**(DDMat a, DDMat d)
static DXMat	**add(DXMat a, DXMat d)**

Two or more matrices of identical dimensions *m* and *n* can be added. Given *m*-by-*n* matrices *A* and *B*, their **sum** *A* + *B* is the *m*-by-*n* matrix computed by adding corresponding matrix elements.

```
double [][] aData = {{1,2,3}, {4,5,6}, {7,8,9}};
double [][] bData = {{5,6,7}, {9,7,8}, {3,2,1}};
DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
DDMat C = DDMat.add(A, B);
```

Matrix Subtraction

static DDMat	**subtraction**(DDMat a, DDMat d)
static DXMat	**subtraction**(DXMat a, DXMat d)

Two or more matrices of identical dimensions *m* and *n* can be subtracted. Given *m*-by-*n* matrices *A* and *B*, their **difference** *A* - *B* is the *m*-by-*n* matrix computed by subtracting corresponding matrix elements.

```
double [][] aData = {{1,2,3}, {4,5,6}, {7,8,9}};
double [][] bData = {{5,6,7}, {9,7,8}, {3,2,1}};
DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
DDMat C = DDMat.subtraction(A, B);
```

Matrix Multiplication

static DDMat	**multiply**(DDMat a, DDMat b) (matrix * matrix) operator overload
static DDMat	**multiply**(DDMat a, double b) (matrix * scalar) operator overload
static DDMat	**multiply**(double b, DDMat a) (scalar * matrix) operator overload

static DXMat	**multiply**(DComplex b, DXMat a) (scalar * matrix) operator overload
static DXMat	**multiply**(double b, DXMat a) (scalar * matrix) operator overload
static DXMat	**multiply**(DXMat a, DComplex b) (matrix * scalar) operator overload
static DXMat	**multiply**(DXMat a, double b) (matrix * scalar) operator overload
static DXMat	**multiply**(DXMat a, DXMat b) (matrix * matrix) operator overload

Two or more matrices of proper dimenson can be multiplied. The number of columns of the left matrix must be the same as the number of rows of the right matrix. Given a matrix A (*m* x *n)* and a matrix B (n x p), the size of the matrix C, that is the product of A * B = C, is (m x p).

```
double [][] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}}; // (3 x 4)
double [][] bData = {{5,6,7,8,9},{9,7,8,6,4},{3,2,1,0,9},{3,7,0,4,7}}; // (4 x 5)
DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
DDMat C = DDMat.multiply(A, B); // (3 x 5)
```

```
double [][] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}}; // (3 x 4)
double [][] bData = {{5,6,7,8,9},{9,7,8,6,4},{3,2,1,0,9},{3,7,0,4,7}}; // (4 x 5)
DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
DDMat C = DDMat.multiply(A, B); // (3 x 5)
```

```
double [][] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}}; // (3 x 4)
DDMat A = new DDMat(aData);
double c = 3.1415;
DDMat B = DDMat.multiply(A, c);
```

```
double [][] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}}; // (3 x 4)
double [][] bData = {{5,6,7,8,9},{9,7,8,6,4},{3,2,1,0,9},{3,7,0,4,7}}; // (4 x 5)
DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
DDMat C = DDMat.multiply(A, B); // (3 x 5)
```

Matrix Division

static DDMat	**divide**(DDMat b, DDMat a) solves the linear system of equations x = b/A, or more appropriatly Ax = b.
static DXMat	**divide**(DXMat b, DXMat a) solves the linear system of equations x = b/A, or more appropriatly Ax = b.

Matrix division is a bit of a misnomer, since the divide sign doesn't often appear in linear algebra forumlas. Take the the matrix expression b/A. This can be rewritten as $x = b*A^{-1}$, where A^{-1} is the matrix inverse of A. In general, you never want to solve for a matrix inverse, then use that inverse in subsequent calculations. It is not as accurate, nor as fast as alternative methods. When this software sees a divide sign in a matrix expression, it redefines the original problem x = b/A as the matrix expression Ax = b, which is the standard format for solving for system of linear equations, solves for x, then replaces the expression b/A with the matrix x. In order for this to work properly, A must be square and the number of rows in b must equal the number of columns in A.

```
double [][] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}, {10,1,5,7}}; // (4 x 4)
double [][] bData = {{5},{6},{7},{8}}; // (4 x 1)
DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
```

```
DDMat C = DDMat.divide(A, B);
```

Scalar Division

static DDMat	**divide**(DDMat a, double b) (matrix / scalar)

static DXMat	**divide**(DXMat a, DComplex b) (matrix / scalar)
static DXMat	**divide**(DXMat a, double b) (matrix / scalar

Given a matrix A, and a scalar c, the division of A / c is calculated by dividing every element of the matrix A by the scalar c.

```
double [][] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}}; // (3 x 4)
DDMat A = new DDMat(aData);
double c = 3.1415;
DDMat B = DDMat.divide(A, c);
```

Mix all of the operators in complex math expressions

```
[C#]
double [,] aData = {{1,2,3,4},{4,5,6,7},{7,8,9, 10}}; // (3 x 4)
double [,] bData = {{5,6,7,8,9},{9,7,8,6,4},{3,2,1,0,9},{3,7,0,4,7}}; // (4 x 5)
double [,] cData = {{5,6,7,8,9},{9,7,8,6,4},{3,2,1,0,9}}; // (3 x 5)

DDMat A = new DDMat(aData);
DDMat B = new DDMat(bData);
DDMat C = new DDMat(cData);

DDMat D = A * B + C * 3.1415; // (3 x 5)

[VB]
Dim aData As Double(,) = {{1, 2, 3, 4}, {4, 5, 6, 7}, {7, 8, 9, 10}}  ' (3 x 4)
Dim bData As Double(,) = {{5, 6, 7, 8, 9}, {9, 7, 8, 6, 4}, {3, 2, 1, 0, 9}, {3,
7, 0, 4, 7}}  ' (4 x 5)
Dim cData As Double(,) = {{5, 6, 7, 8, 9}, {9, 7, 8, 6, 4}, {3, 2, 1, 0, 9}}  '3 x
5
Dim A As New DDMat(aData)
Dim B As New DDMat(bData)
Dim C As New DDMat(cData)
```

```
'   Dim D As DDMat = A * B + C * 3.1415 ' (3 x 5) VB 2005 and greater only
Dim D As DDMat = DDMat.Add(DDMat.Multiply(A, B), DDMat.Multiply(C, 3.1415))
```

Matrix Methods

Many methods are available in the **DDMat** and **DXMat** classes that manipulate the rows and columns of the matrix. Other methods calculate important characteristics of the matrix. Since the **DDMat** and **DXMat** classes include the same routines they are documented in parallel. A subset of the available methods follows. A complete description is found in the XXXXX help file.

Method DDMat.C, DXMat.C

Returns the specified column, or a sub range of the column, as a vector

DDMat

DDMat	**C**(int col) Returns the specified column as a vector.
DDMat	**C**(int col, int start, int num) Returns a sub-range of the specified column as a vector.

DXMat

DXMat	**C**(int col) Returns the specified column as a vector.
DXMat	**C**(int col, int start, int num) Returns a sub-range of the specified column as a vector.

Parameters

col specified column
start specified row to start copying from
start number to copy

Return Value

Returns the specified column, or a sub range of the column, as a vector

Method DDMat.clone, DXMat.clone

Returns an object that is a clone of this array object.

DDMat

java.lang.Object	**clone**() .

DXMat

java.lang.Object	**clone**()

Return Value

Returns a clone of this matrix object.

Method DDMat.copy, DXMat.copy

Copy a matrix.

DDMat

void	**copy**(DMatBase src)

DXMat

void	**copy**(XMatBase src)

Parameters

src source array

Method DDMat.determinant, DXMat.determinant, DDMat.D, DXMat.D

Returns the determinant of the current matrix.

DDMat

double	**D**() Returns the determinant of the current matrix.
double	**determinant**() Returns the determinant of the current matrix.
static double	**determinant**(DDMat A) Returns the inversion, and determinant of the specified matrix.

DXMat

DComplex	**D**()
DComplex	**determinant**()
static DComplex	**determinant**(DXMat A)

Return Value

Returns the determinant.

Method DDMat. equals, DXMat.equals

Equals comparator. Two matrices are considered equal if they are the same dimension and if each corresponding element at matrix index [i,j] is equal.

DDMat

boolean	**equals**(DMatBase obj)

DXMat

boolean	**equals**(XMatBase obj)

Parameters

obj Returns true if the source object is equal to the current object.

Return Value

Returns true if the source object is equal to the current object.

Method DDMat.getDataBuffer, DXMat.getDataBuffer

Returns a reference to the internal **DComplex** [][] data buffer that stores the matrix data.

DDMat

double[][]	**getDataBuffer**()

DXMat

DComplex[][]	**getDataBuffer**()

Return Value

Returns a reference to the internal matrix buffer.

Method DDMat.getDiagonal, DXMat.getDiagonal

Returns a vector that is the diagonal of the matrix.

DDMat

DDMat	**getDiagonal**()

DXMat

DXMat	**getDiagonal**()

Return Value

Returns a vector that is the diagonal of the matrix.

Method DDMat.getElement, DXMat.getElement

Returns the element of the matrix.

DDMat

double	**getElement**(int element) Set an element of the vector.
double	getElement(int row, int column) **Returns the element of the matrix.**

DXMat

double	**getElement**(int element) Set an element of the vector.
double	getElement(int row, int column) **Returns the element of the matrix.**

Parameters

row The row of the matrix element.
column The column of the matrix element.

Return Value

Returns the value of the matrix element.

Method DDMat.getMatrix, DXMat.getMatrix

Initialize an external matrix to match this one.

DDMat

void	**getMatrix**(DDMat dest)

DXMat

void	**getMatrix**(DXMat dest)

Parameters

dest destination array

Method DDMat.getTranspose, DXMat.getTranspose

Returns the transpose of a matrix. A complex transpose converts the complex numbers to their complex conjugate: i.e, a + bi -> a - bi.

DXMat

DDMat	**getTranspose**()

DXMat

DDMat	**getTranspose**()

Return Value

Returns the transpose of the matrix.

Method DDMat.getVector, DXMat.getVector

Returns a [] vector of the data in the matrix.

DDMat

double[]	**getVector**()

DXMat

DComplex[]	**getVector**()

Return Value

Returns a [] vector of the data in the matrix.

Method DDMat.getVectorElement, DXMat.getVectorElement

Get an element of a vector.

DDMat

double	**getVectorElement**(int element)

DXMat

DComplex	**getVectorElement**(int element)

Parameters

element The index of the vector element.

Return Value

The value of the vector element.

Method DDMat.I, DXMat.I

Returns the inversion of the current matrix.

DDMat

DDMat	**I**()

DXMat

DXMat	**I**()

Return Value

Returns the inversion of the current matrix.

Method DDMat.invert, DXMat.invert

Returns the inversion of the specified matrix.

DDMat

static int	**invert**(DDMat A)

DXMat

static int	**invert**(DXMat A)

Parameters

A Returns the inversion of the matrix, overwriting the original matrix.

Return Value

Returns an error code. See the **MatError** class error code list.

Method DDMat.makeRandomLEq, DXMat.makeRandomLEq

Make a pair of matrices that form the A and b matrices of a linear set of equations, Ax = b. The matrices are constructed so that the solution (x) of Ax = b is always a vector with the values [1, 2, 3, ..., rank].

DDMat

static void	**makeRandomLEq**(DDMat A, DDMat b, int rank, boolean posdefsym)

DXMat

static void	**makeRandomLEq**(DXMat A, DXMat b, int rank, boolean posdefsym)

Parameters

A	Returns the coefficient array, A, of the linear system of equations.
b	Returns the right hand side array, b, of the linear system of equations.
rank	The number of rows and columns of the coefficient A matrix.
posdefsym	Set to true and the A matrix will be symmetric (hermitian)-positive-definite

Method DDMat.makeRandomMatrix, DXMat.makeRandomMatrix

Make a random, square, matrix of the specified rank and symmetry.

DDMat

`static DDMat`	**`makeRandomMatrix`**`(int rank, boolean posdefsym)`

DXMat

`static DXMat`	**`makeRandomMatrix`**`(int rank, boolean posdefsym)`

Parameters

rank	The number of rows and columns of the generated matrix.
posdefsym	Set to true and the A matrix will be symmetric (hermitian for complex) - positive-definite

Return Value

Returns the new matrix.

Method DDMat.makeRHSFromCoefs, DXMat.makeRHSFromCoefs

Using the coefficient matrix A, a rhs vector (b) is constructed so that the solution x, of the linear system of equations Ax = b is [1, 2, 3, ..., rank]

DDMat

`static DDMat`	**`makeRHSFromCoefs`**`(DDMat A)`

DXMat

[

`static DXMat`	**`makeRHSFromCoefs`**`(DXMat A)`

Parameters

A	Returns the coefficient array, A, of the linear system of equations.

Return Value

Returns the rhs vector.

Method DDMat.product, DXMat.product

Calculates the product of two matrices (A * B) and returns the result as a third matrix.

DDMat

`static DDMat`	**`product`**`(DDMat A, DDMat B)`

DXMat

`static DXMat`	**`product`**`(DXMat A, DXMat B)`

Parameters

A The first of two arguments in the matrix multiple (A * B).
B The second of two arguments in the matrix multiple (A * B).

Return Value

Returns the product of two matrices (A * B).

Method DDMat.R, DXMat.R

Returns the specified row, or a sub range of the row, as a vector

DDMat

`DDMat`	**`R`**`(int row)`
`DDMat`	**`R`**`(int row, int start, int num)`

DXMat

`DXMat`	**`R`**`(int row)`
`DXMat`	**`R`**`(int row, int start, int num)`

Parameters

row specified column
start specified column to start copying from
start number to copy

Return Value
Returns the specified row, or a sub range of the row, as a vector

Method DDMat.setElement, DXMat.setElement

Set a matrix element.

DDMat

void	**setElement**(int element, double value)
void	**setElement**(int row, int column, double value)

DXMat

void	**setElement**(int element, DComplex value)
void	**setElement**(int row, int column, DComplex value)

Parameters

row The row of the matrix element.
column The column of the matrix element.
value The value of the matrix element.

Method DDMat.setMatrix, DXMat.setMatrix

DDMat

void	**setMatrix**(double[][] source)
void	**setMatrix**(DDMat source)

DXMat

void	**setMatrix**(DComplex[][] source) High speed routine for copying a matrix.
void	**setMatrix**(DXMat source) initialize a matrix using a DXMat source.

Parameters

source source array

Method DDMat.T, DXMat.T

Returns the transpose of the current matrix. A complex transpose converts the complex numbers to their complex conjugate: i.e., a + bi -> a - bi.

DDMat

DDMat	**T**()

DXMat

DXMat	**T**()

Return Value

Returns the transpose of the current matrix.

Method DMatBase.load, XMatBase.load

These method will open a text file and read the CSV (Comma Separated Value) formatted data values in that file. A CSV file can be created by popular spreadsheet and word processing programs. Some localization for different operating systems and locales can be handled by the modifying the default *csv* (**MatCSV**) object. The file can be organized so that the columns represent groups and the rows represent data values for each group (COLUMN_MAJOR), or the where the rows represent groups and the columns represent data values for each group (ROW_MAJOR). Use the **MatCSV.setOrientation** method to initialize the *csv* argument for the proper data orientation.

int	**load**(MatCSV csv, java.lang.String filename, int rowskip, int columnskip)
int	**load**(java.lang.String filename)

Parameters

csv	An instance of a MatCSV object.
filename	The name of the file.
rowskip	Skip this many rows before starting the read operation
columnskip	For each new row, skip this many columns before starting read.

Method DMatBase.save, XMatBase.save

This method will create a text file and output the dataset to that file in a CSV (Comma Separated Value) format. A CSV file can be read by popular spreadsheet and word processing programs. Some localization for different operating systems and locales can be handled by the modifying the default *csv* (MatCSV) object. The file can be organized so that the columns represent groups and the rows represent data values for each group (COLUMN_MAJOR), or the where the rows represent groups and the columns represent data values for each group (ROW_MAJOR). Use the MatCSV.setOrientation method to initialize the *csv* argument for the proper data orientation.

int	**save**(MatCSV csv, java.lang.String filename)
int	**save**(MatCSV csv, java.lang.String filename, boolean append)
int	**save**(java.lang.String filename)

Parameters

csv	An instance of a MatCSV object.
filename	The name of the file.
append	True and if the file exists, data is appended to the file.

A Summary of all DDMat, DXMat Methods

Public Static (Shared) Methods

determinant	Overloaded. Returns the inversion, and determinant of the specified matrix.
invert	Returns the inversion of the specified matrix.
makeRandomLEq	Make a pair of matrices that form the A and b matrices of a linear set of equations, Ax = b. The matrices are constructed so that the solution (x) of Ax = b is always a vector with the values [1, 2, 3, ...,rank].
makeRandomMatrix	Make a random, square, matrix of the specified rank, symmetry and diagonal dominance.
makeRHSFromCoefs	Using the coefficient matrix A, a rhs vector (b) is constructed so that the solution x, of the linear system of equations Ax = b is [1, 2, 3, ..., rank]
product	Calculates the product of two matrices (A * B) and returns the result as a third matrix.

Public Instance Constructors

DDMat	Overloaded. Initializes a new instance of the DDMat class.

Public Instance Properties

getDataBuffer	Returns a reference to the Array object that represents internal data buffer. This reference will remain valid only as long as the array is not forced to resize, forcing a reallocation of the internal buffer.
isSquare	Returns true if the matrix is square (numRows == numColumns).
isSymmetric	Returns true if the matrix is symmetric (A = Transpose(A)).
getLength	Returns the number of rows * columns in the matrix.
getNumColumns	Returns the number of columns in the matrix.
getNumRows	Returns the number of rows in the matrix.
isOK	Returns true if no error pending in matrix.

Public Instance Methods

addConstCol	Add a constant to a column of the current matrix.
addConstRow	Add a constant to a row of the current matrix.
C	Overloaded. Returns the specified column as a vector.
cholesky_Solve	Solve a system of linear equations (using Cholesky Decomposition) in the standard form Ax = b, where A is the coefficient matrix, b is the right hand side and x is the solution. The A matrix must be symmetric positive-definite matrix. Cholesky is the fastest technique for solving linear systems of equations, though it does require the coefficient matrix be symmetric positive-definite.
clone	Returns an object that is a clone of this matrix object.
getCondNum	Calculate square matrix condition number
convertFromBaseOne	Convert the current matrix from a base one matrix to a base zero matrix.

convertToBaseOne	Convert a base zero matrix to a base one matrix.
copy	Copy a matrix.
D	Returns the determinant of the current matrix.
deleteCol	Delete a column from the current matrix, decreasing the column count of the matrix by one.
deleteRow	Delete a row from the current matrix, decreasing the row count of the matrix by one.
deter	Returns the determinant of the current matrix. Gauss-Jordan is used.
determinant	Overloaded. Returns the determinant of the current matrix.
dimMatrix	Dimension the matrix for size [0..rows-1, 0..columns-1]
dimZero	Returns true if the matrix dimensioned to 0,0.
Equals	Overloaded. Determines whether the specified Object is equal to the current **Object**.
equals	Overloaded. Determines whether the specified Object is equal to the current **Object**.
freeMat	Initializes the matrix to empty, 0 rows and 0 columns.
fromDMatBase	Initializes the matrix from the source matrix.
fromTriplets	Initialize the matrix using an array of triplets (DTriplet). Elements not initialized remain at their default 0.0 value.
getBaseOneMatrix	Return a copy of the current matrix as a base one array.
getCol	Get a 1D array representing a copy of a column in this matrix.
getCompatibleMatrix	Overloaded. Get a matrix compatible with this one, initialized with and dimensioned to the size of the source array.
getDataBuffer	Returns a reference to the internal double [][] data buffer that stores the matrix data.
getDiagonal	Overloaded. Get a vector representing a copy of the diagonal elements of this matrix.
getDiagonal	Overloaded. Returns a vector that is the diagonal of this one.

getElement	Returns the element of the matrix.
getError	Return the current matrix error.
getLength	Returns the number of rows * columns in the matrix.
getMatrix	Overloaded. Get a 2D array representing a copy of the values in this matrix.
getMatrix	Overloaded. Initialize an external DDMat matrix to match this one.
getNumNonZeroEl	Returns the number of non-zero elements in the matrix.
getRC	Get a row or a column of the matrix.
getRow	Get a row in the matrix.
getSubCol	Get a sub range of a column in the matrix.
getSubRC	Get a sub range of a row or a column in the matrix.
getSubRow	Get a sub range of a row in the matrix.
getTransp	Returns the transpose of the current matrix.
getTranspose	Returns a matrix that is the transpose of this one.
getTriplets	Get an array of triplets representing the non-zero elements of the matrix.
getVector	Returns a double [] vector of the data in the matrix.
getVectorElement	Set an element of the vector.
gJ_Solve	Solve a system of linear equations (using Gauss-Jordan) in the standard form Ax = b, where A is the coefficient matrix, b is the right hand side and x is the solution. The A matrix must be symmetric positive-definite matrix. Gauss-Jordan with partial pivoting is a very stable method of solving systems of equations.
I	Returns the inversion of the current matrix.
initialize	Initialize the entire matrix to the specified value.

insertCol	Insert a column into the current matrix, increasing the column count of the matrix by one.
insertRow	Insert a row into the current matrix, increasing the row count of the matrix by one.
inv	Returns the inverse of the array. Gauss-Jordan is used.
isNonZero	Returns true if a matrix element is non-zero.
lES_Solve	Solve a system of linear equations in the standard form Ax = b, where A is the coefficient matrix, b is the right hand side and x is the solution. If A is a square matrix, then LU Decomposition is used to solve for x. If A.NumRows > A.NumColumns, i.e. the system of equations is over determined, the QR algorithm is used to solve for x.
load	Overloaded. These methods will open a text file and read the CSV (Comma Separated Value) formatted data values in that file. A CSV file can be created by popular spreadsheet and word processing programs. This version of Load assumes CSV values are organized in a ROW_MAJOR format.
lU_Solve	Solve a system of linear equations (using LU Decomposition) in the standard form Ax = b, where A is the coefficient matrix, b is the right hand side and x is the solution.
makeHilbert	Make a Hilbert matrix of the specified rank.
matNull	Initializes the matrix to empty, 0 rows and 0 columns.
norm	Calculate the a norm for the current matrix.
norm1	Calculate the 1-norm of the current matrix.
norm2	Calculate the 2-norm of the current matrix.
normInf	Calculate the infinity-norm of the current matrix.
numElRC	Returns the number of row or column elements in the matrix.
qR_Solve	Solve a system of linear equations (using QR Factorization) in the standard form Ax = b, where A is the coefficient matrix, b is the right hand side and x is the solution.
R	Overloaded. Returns the specified row as a vector.
rCInitialize	Initialize a row or a column to the specified value.

rCNorm	Returns the vector norm for a row or a column in the matrix.
rCSSQ	Returns the sum of squares for a row or a column in the current matrix.
rCSubInitialize	Initialize a sub-range of a row or a column to the specified value.
reset	Initializes the matrix to empty, 0 rows and 0 columns.
resize	Overloaded. Resize the vector. This is a non-destructive resize. Original data, to the extent that it fits into the new vector, is preserved.
save	Overloaded. These methods will create a text file and output the dataset to that file in a CSV (Comma Separated Value) format. A CSV file can be read by popular spreadsheet and word processing programs. This version of Save assumes CSV values are organized in a ROW_MAJOR format.
scaleCol	Scale a column in the current matrix.
scaleMat	Scale all of the elements of the current matrix
scaleRow	Scale a row in the current matrix.
setBaseOneMatrix	Set the current matrix as using a base one matrix.
setCol	Overloaded. Set a column in the current matrix.
setElement	Set an element of the matrix.
setError	Set the current matrix error.
setMatrix	Overloaded. Initializes the vector using the specified 1D array. Source array.
setMatrix	Overloaded. High speed routine for copying a matrix. Uses System.Array.Copy.
setRC	Overloaded. Set a row or a column.
setRow	Overloaded. Set a row in the current matrix.
setSize	Overloaded. Set the size of the matrix.
setSubCol	Overloaded. Set a sub-range of a column in the current matrix.

setSubRC	Overloaded. Set the sub-range of a row or a column.
setSubRow	Overloaded. Set a sub-range of a row in the current matrix.
setVector	Set the values of a matrix vector using a double array.
setVectorElement	Set an element of the vector.
sSQ	Returns the sum of squares of all elements in the matrix.
sum	Returns the sum of all elements in the matrix.
sumAbs	Returns the sum of the ABS of all elements in the matrix.
swapCols	Swap a pair of columns of the current matrix.
swapRows	Swap a pair of rows of the current matrix.
T	Returns the transpose of the current matrix.
toString	Returns a String that represents the current Object.
transp	Transpose the current matrix.

Chapter 5 - Solution of Linear Systems of Equations

LESBase
- **LESLU**
- **LESGJ**
- **LESCholesky**
- **LESQR**

LESXBase
- **LESXLU**
- **LESXGJ**
- **LESXCholesky**
- **LESXQR**

The central problem of linear algebra and one of the most common numerical problems encountered is the solution of a set of linear equations. Typically, this takes the form of equations that when fully written out look something like the equations below.

$$\begin{aligned} 3x_1 &+ 2x_2 - x_3 = 1 \\ 2x_1 &- 2x_2 + 4x_3 = -2 \\ -x_1 &+ \tfrac{1}{2}x_2 - x_3 = 0 \end{aligned}$$

The coefficients of the left hand side are the numbers in front of the x1, x2 and x3 variables and are usually referred to as the coefficient matrix variable A. The numbers of the right hand side, 1, -2, 0 in this case, are usually referred to as the *rhs*, or the matrix variable b. The x's, or unknowns, are referred to using the matrix variable x. When written in matrix format, the system of equations above is written as:

Ax = b

where,

A = $\begin{bmatrix} 3 & 2 & -1 \\ 2 & -2 & 4 \\ -1 & 0.5 & -1 \end{bmatrix}$ and **b** = $\begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$

The solution, x, to this particular set of equations is:

x = $\begin{bmatrix} 1 \\ -2 \\ -2 \end{bmatrix}$

The matrix representation is generalized for systems of equations of any size, whether 2 x 2 or 10000 x 10000. The real number example above can be further generalized to the domain of complex numbers.

Solving large systems of equations is a critical element in almost every engineering discipline. Typical applications include circuit analysis, finite element analysis, digital signal processing, forecasting, simulation, econometrics, linear programming, robotics, computer animation, genetic engineering, nuclear engineering, etc. The list is endless.

The first formal method for solving systems of simultaneous equations is usually attributed to Frederick Gauss, the 19th century German mathematician. His general algorithm was further improved over the years and appears here in the numerically most stable form, Gauss-Jordan with partial pivoting. Other algorithms included in the software include LU Decomposition, QR Factorization (or Decomposition), and Cholesky Decomposition. Each algorithm has specific benefits, and weaknesses, you need to consider when deciding which algorithm to use. Each algorithm is supplied in versions for both real and complex numbers.

Algorithm	**Strengths**	**Weaknesses**
Gauss-Jordan	Stable for all but the most ill-conditioned of matrices. Produces the inverse of the coefficient matrix (A) as a by-product. Produces a solution in finite time on the order of $2n^3 / 3$ operations.	The slowest of the four algorithms presented here. Since it does not produce an intermediate matrix factorization, it is not the recommended algorithm if you need to be able to solve multiple right hand sides (b) for the same coefficient matrix (A). Requires a square coefficient matrix (A).
LU	Stable for all but the most ill-conditioned of matrices. Faster than Gauss-Jordan when used to solve for one right hand side. Produces an intermediate matrix factorization that can be used for high-speed solution of multiple right hand sides using back-substitution. Produces a solution in finite time on the order of $2n^3 / 3$ operations.	Not as fast as Cholesky decomposition in solving symmetric, positive, definite matrices. Requires a square coefficient matrix (A).
QR	Stable for all but the most ill-conditioned of matrices. Faster than Gauss-Jordan when used to	Not as fast as Cholesky decomposition in solving symmetric,

	solve for one right hand side. Produces an intermediate matrix factorization that can be used for high-speed solution of multiple right hand sides using back-substitution. Can be used for solving over-determined systems of equations, where the number of rows is greater than the number of columns. Produces a solution in finite time on the order of $2n^3 / 3$ operations.	positive, definite matrices.
Cholesky	The fastest of the four algorithms. Produces an intermediate matrix factorization that can be used for high-speed solution of multiple right hand sides using back-substitution. Produces a solution in finite time on the order of $n^3 / 3$ operations, or about twice as fast as LU decomposition.	Requires that the coefficient matrix is a symmetric, positive, definite matrix.

Standard Interface for the Solution of Linear Equations

The linear equations solving classes all use the same interface. You define the problem by passing the coefficient matrix A in the constructor of the class you are using. In the case of LU, QR and Cholesky, the coefficient matrix is immediately factored into intermediate matrices. The programmer then calls the **solve** method, passing in a matrix of one or more right hand sides (b). The solution is calculated using the intermediate matrices and back substitution and returned in the same **solve** method call. The Gauss-Jordan algorithm does not factor the coefficient matrix into intermediate matrices, and the 100% of the algorithm takes place when the s**olve** method is called. The **inverse** method returns the coefficient matrix inverse. The **checkSolution** method will check the accuracy of a solution by calculating right hand side values based on the original coefficient matrix and solution, and comparing the results to the actual right hand side values.

The problem presented in the first page of the chapter is solved below using each of the four algorithms.

```
double [][] aData = {{3,2,-1},{2,-2,4},{-1, 0.5, -1}};
double [] bData = {1, -2, 0};
```

```
DDMat A = new DDMat(aData);
DDMat b = new DDMat(bData);

DDMat x = new DDMat();
A.lES_Solve(b,x);



// LU Decomposition
LESLU leslu = new LESLU(A);
leslu.solve(b,x);

// Gauss-Jordan
LESGJ lesgj = new LESGJ (A);
lesgj.solve(b,x);

// QR Factorization
LESQR lesqr = new LESQR (A);
lesqr.solve(b,x);
```

While the Cholesky algorithm is used in exactly the same way, it will fail for the example above because the coefficient matrix (A) is not symmetric-positive-definite. A symmetric, positive definite matrix is used in the example below.

```
[C#]
// Cholesky Decomposition
// Cholesky requires a symmetric-positive-definite matrix
double [][] aSPDData = {{ 13,   2,  -1},
                        { 2, 22,    4},
                        {-1,   4,   15}};
DDMat ASPD = new DDMat(aSPDData);
LESCholesky leschol = new LESCholesky (ASPD);
leschol.solve(b,x);
```

The complex versions of the same algorithms follow exactly the same logic, except that **DComplex** objects are used instead of doubles, and complex **DXMat** matrices are used instead of real **DDMat** matrices.

```
double [][] arData = {{1.6,2.9,8.7}, {5.6,6.1,7.4}, {4.5, 4.8, 3.2}};
double [][] aiData = {{8.7,2.2,6.1}, {7.4,2.5,3.4}, {5.4, 8.1, 3.0}};

double [] brData = {0.0, 20, -10};
```

```
double [] biData = {60, 60, 50};

DXMat A = new DXMat(arData, aiData);
DXMat b = new DXMat(brData, biData);

DXMat x = new DXMat();

A.lES_Solve(b,x);

// QR Factorization
LESXQR lesqr = new LESXQR (A);
lesqr.solve(b,x);



// LU Decomposition
LESXLU leslu = new LESXLU(A);
leslu.solve(b,x);

// Gauss-Jordan
LESXGJ lesgj = new LESXGJ (A);
lesgj.solve(b,x);

double [][] aSPDDataR = {{ 13,   2,  -1},
                { 2, 22,    4},
                {-1,   4,   15}};
double [][] aSPDDataI = {{ 0,   2,  -1},
                { -2, 0,    4},
                {1,   -4,   0}};
DXMat ASPD = new DXMat(aSPDDataR, aSPDDataI);
LESXCholesky leschol = new LESXCholesky(ASPD);
b = DXMat.makeRHSFromCoefs(ASPD);
leschol.solve(b,x);;
```

Gauss-Jordan

Class LESGJ, LESXGJ

The **LESGJ**, and **LESXGJ** classes implement the Gauss-Jordan with partial pivoting algorithm, the most stable version of the Gauss-Jordan algorithm. Gaussian elimination turns the original, square, non-singular, system of equations into one that is upper triangular using a technique of subtracting multiples of one equation from another. Once the system of equations is upper triangular, the solution is solved for using simple back substitution. The partial pivoting enhancement of the algorithm ensures that extremely

small values are not selected as equation multipliers, which can introduce round-off errors that lower the numerical accuracy of the solution. The algorithm produces the inverse of the original coefficient matrix as a by-product.

LESGJ, LESXGJ Constructors

LESGJ

```
public LESGJ(   DMatBase a);
```

LESXGJ

```
public LESGJ(   XMatBase a);
```

Parameters

a The coefficient matrix, A, for the Ax = b system of equations.

Example

The code in the **Standard Interface for the Solution of Linear Equations** section is extracted from the SimpleLES and SimpleLESX example programs. The class is also used in the LESTest and LESXTest example programs.

LU Decomposition

Class LESLU, LESXLU

The **LESLU**, and **LESXLU** classes implement the Crout version of the LU decomposition algorithm. LU decomposition starts with the standard definition of a linear system of equations, Ax = b, where A is square (n equations for n unknowns). It factors the non-singular matrix A into the matrix product, A = LU, of a lower triangular matrix L and upper triangular matrix U with unit diagonal. The resulting linear algebra equation LUx = b is solved for x. Most of the work is done in the factoring or A into the L and U matrices. Once L and U are calculated, for a given right hand side, b, the solution x, can be quickly calculated using a combination of forward and back substitution. For a single right hand side, the LU decomposition algorithm is two to three times faster than Gauss-Jordan algorithm. Since the LU matrices do not need to be recalculated for each new right hand side, it is even faster if you are solving for multiple right hand sides.

Unlike the Gauss-Jordan algorithm, the LU algorithm does not produce a matrix inverse as a by-product. If you call the **Inverse** method, the inverse is calculated by solving the Ax = b equation using b (b is a square matrix in this case with the same dimensions as A) as the identity matrix. If Ax = b, and b is the identity matrix, then x must be the inverse of A. Solving for x produces the inverse of A.

LESLU, LESXLU Constructors

LESLU

```
public LESLU(   DMatBase a);
```

LESXLU

```
public LESLU(   XMatBase a);
```

Parameters

a The coefficient matrix, A, for the Ax = b system of equations.

Example

The code in the **Standard Interface for the Solution of Linear Equations** section is extracted from the SimpleLES and SimpleLESX example programs. The class is also used in the LESTest and LESXTest example programs.

QR Factorization

Class LESQR, LESXQR

The **LESQR**, and **LESXQR** classes implement the QR factorization by way of a Householder reflections algorithm. QR factorization starts with the standard definition of a linear system of equations, Ax = b. Unlike Gauss-Jordan, LU or Cholesky algorithms, the A matrix does not have to be square. The system of equations can be over-determined, where the number of equations is greater than the number of unknowns, producing an m x n matrix (number of equations x number of unknowns) with m >= n. The algorithm factors the non-singular matrix A into the matrix product A = QR of an orthogonal matrix Q and upper triangular matrix R. The resulting linear algebra equation, QRx = b, is solved for x. Most of the work is done in the factoring or A into the Q and R matrices. Once Q and R are calculated, for a given right hand side, b, the solution x, can be quickly calculated using a combination of forward and back substitution. For a single right hand side, the QR decomposition algorithm is two to three times faster than Gauss-Jordan algorithm. Since the QR matrices do not need to be recalculated for each new right hand side, it is even faster if you are solving for multiple right hand sides.

Unlike the Gauss-Jordan algorithm, the QR algorithm does not produce a matrix inverse as a by-product. If you call the **Inverse** method, the inverse is calculated by solving the Ax = b equation using b (b is a square matrix in this case with the A dimensions (n x n) as the identity matrix. If Ax = b, and b is the identity matrix, then x must be the inverse of A. Solving for x produces the inverse of A.

When the QR algorithm is solving for the over determined case where the number of equations exceeds the number of unknowns, the result is the same as a multiple regression problem run on the same data. The solution are the values that minimize the sum of the residual errors squared, usually referred to as the linear least squares solution to the problem.

LESQR, LESXQR Constructors

LESQR

```
public LESQR(   DMatBase a);
```

LESXQR

```
public LESQR(   XMatBase a);
```

Parameters

a The coefficient matrix, A, for the Ax = b system of equations.

Example

The code in the **Standard Interface for the Solution of Linear Equations** section is extracted from the SimpleLES and SimpleLESX example programs. The class is also used in the LESTest and LESXTest example programs.

Cholesky Decomposition

Class LESCholesky, LESXCholesky

The **LESCholesky**, and **LESXCholesky** classes implement the Cholesky decomposition algorithm. Cholesky decomposition starts with the standard definition of a linear system of equations, Ax = b. The coefficient matrix A must be symmetric-positive-definite in order for the algorithm to work. The Cholesky algorithm factors the non-singular matrix A into the matrix product of an upper and lower triangular matrix A = LU (much like the LU algorithm). In this case, L and U are specially factored so that it U is the transpose of L. So the equation becomes $A = LL^*$. The resulting linear algebra equation $LL^*x = b$ is solved for x. Most of the work is done in the factoring of the L matrix. Once L is calculated, for a given right hand side, b, the solution x, can be quickly calculated using a combination of forward and back substitution. For a single right hand side, the Cholesky decomposition algorithm is two to three times faster than the LU algorithm. Since the Cholesky L matrix does not need to be recalculated for each new right hand side, it is even faster if you are solving for multiple right hand sides.

Unlike the Gauss-Jordan algorithm, the Cholesky algorithm does not produce a matrix inverse as a by-product. If you call the **Inverse** method, the inverse is calculated by solving the Ax = b equation using b (b is a square matrix in this case with the same dimensions as A) as the identity matrix. If Ax = b, and b is the identity matrix, then x must be the inverse of A. Solving for x produces the inverse of A.

In the complex version of the algorithm (**LESXCholesky**), the coefficient matrix, A, must be Hermitian-positive-definite, since Hermitian is the complex equivalent of the real symmetric. Since L is complex, L^* is the conjugate transpose of L.

LESCholesky, LESXCholesky Constructors

LESCholesky

```
public LESCholesky (   DMatBase a);
```

LESXCholesky

```
public LESCholesky (   XMatBase a);
```

Parameters

a The coefficient matrix, A, for the Ax = b system of equations.

Example

The code in the **Standard Interface for the Solution of Linear Equations** section is extracted from the SimpleLES and SimpleLESX example programs. The class is also used in the LESTest and LESXTest example programs.

Selected LESGJ, LESXGJ Methods

LESGJ.solve, LESXGJ.solve Methods

Solve the square, non-singular, system of equations represented by the matrix equation Ax = b using the Gauss-Jordan with partial pivoting algorithm. If the right hand side matrix, b, has more than one column, a solution for each column is returned in the columns of the solution matrix x.

LESGJ

```
public int gJ_Solve( DMatBase b,  DMatBase x);
```

LESXGJ

```
public int gJ_Solve( XMatBase b,  XMatBase x);
```

Parameters

b The right hand side matrix, b, for the system of equations Ax = b. The right hand side values of b are stored as a column vector. If b has more then one column, representing multiple right hand sides, the solution, x, will have more than one column, one column for each solution.

x Returns the solution(s), x, to the system Ax = b. If b has more then one column, representing multiple right hand sides, the solution, x, will have more than one column, one column for each solution.

Example

The code in the **Standard Interface for the Solution of Linear Equations** section is extracted from the SimpleLES and SimpleLESX example programs. The class is also used in the LESTest and LESXTest example programs.

Chapter 6 – Eigenvalue and Eigenvector Solutions

EigenBase
- **EigenQR**
- **EigenQL**
- **EigenQR2**

EigenXBase
- **EigenXLR**

Eigenvalues and eigenvectors provide very important tools for analysis of many physical systems. The eigenvalue problem is to find the non-trivial solutions of the equation:

$$\mathbf{AX = \lambda X}$$

where A is a n x n matrix, X is a vector of length n, and λ is a scalar. This equation involves two unknowns λ and X. The n values of λ that satisfy the equation are the eigenvalues of matrix A, and the corresponding values of X are its right eigenvectors. For example:

$$A \quad X \quad = \quad \lambda \quad X$$

$$\begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -2 \\ 2 \end{bmatrix} = 2 \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

The eigenvalue λ in this case is 2 and the eigenvector associated with λ is

$$\begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

To find the eigenvalues of a matrix A, rewrite the equation $AX = \lambda X$ as

$$\begin{aligned} AX &= \lambda IX \\ AX - \lambda IX &= 0 \\ (A - \lambda I)X &= 0 \end{aligned}$$

We want to find the numbers for λ for which the equation above has non-zero solutions X, which is the same as finding the numbers for that make $\det(A - \lambda I) = 0$. This is referred to as the characteristic equation of A. The left hand side of the equation is the characteristic polynomial of A. For the matrix:

$$A = \begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix}$$

$$A - \lambda I = \begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 & 0 \\ 0 & \lambda & 0 & 0 \\ 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & \lambda \end{bmatrix} = \begin{bmatrix} 5-\lambda & 4 & 1 & 1 \\ 4 & 5-\lambda & 1 & 1 \\ 1 & 1 & 4-\lambda & 2 \\ 1 & 1 & 2 & 4-\lambda \end{bmatrix}$$

The characteristic equation $\det(A - \lambda I) = 0$ can be written as follows:

$$\det\begin{bmatrix} 5-\lambda & 4 & 1 & 1 \\ 4 & 5-\lambda & 1 & 1 \\ 1 & 1 & 4-\lambda & 2 \\ 1 & 1 & 2 & 4-\lambda \end{bmatrix} = 0$$

Solving for the determinant will yield the following eigenvalues and their associated eigenvectors:

$$1\begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad 2\begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}, \quad 5\begin{bmatrix} -0.5 \\ -0.5 \\ 1 \\ 1 \end{bmatrix}, \quad 10\begin{bmatrix} 1 \\ 1 \\ 0.5 \\ 0.5 \end{bmatrix}$$

Eigensystem problems are divided into different categories, depending on the symmetry of the original matrix, and whether or not the original matrix contains complex values. The class you should use depends on the category.

	Eigenvalues	**Eigenvectors**
	Class to Use	**Class to Use**
Source Matrix	**(solution numeric type)**	**(solution numeric type)**
Real-Symmetric	EigenQL	EigenQL
	(real)	(real)
Real-Unsymmetric	EigenQR	EigenQR2
	(complex)	(complex)
Complex Hermitian	EigenXLR	EigenXLR
	(complex)	(complex)
Complex Un-Hermitian	EigenXLR	EigenXLR
	(complex)	(complex)

If you have a real-symmetric matrix, use the **EigenQL** class for both eigenvalues and eigenvectors. If you have a real-unsymmetric matrix, use **EigenQR** for eigenvalues only and **EigenQR2** if you want eigenvalues and eigenvectors. If you have a complex matrix, of any type, use the **EigenXLR** class for both eigenvalues and eigenvectors.

Standard Interface for Eigenvalue and Eigenvectors Solutions

The eigen-problem classes use the same interface, with minor variants depending on whether or not the source matrix is real or complex, and whether or not the eigenvalues and eigenvectors are real or complex. You define the problem by passing the coefficient matrix A in the constructor of the class you are using. Real matrices of type **DDMat** use the **EigenQL**, **EigenQR** and **EigenQR2** classes. Complex matrices of type **DXMat** use the **EigenXLR** class. If you need only the eigenvalues of a matrix, use the c**alcEigenValues** method. If you need the eigenvalues and eigenvectors of a matrix, use the c**alcEigenValuesAndVectors**. The c**heckEigenVectors** method will check the accuracy of the solution by comparing the values $AX = \lambda X$.

QL Algorithm for Real Symmetric Matrices

Class EigenQL

The **EigenQL** class will calculate all of the eigenvalues and eigenvectors of a real-symmetric matrix using the implicit QL method.

EigenQL Constructor

```
public EigenQL(  DMatBase a );
```

Parameters

a The source matrix for the eigensystem calculations. The source matrix MUST be real-symmetric.

Method calcEigenValues

Calculate the real eigenvalues for the source matrix, A.

```
public int calcEigenValues( DMatBase eigval);
```

Parameters

eigval Returns the real eigenvalues of the source matrix A.

Return Value

Returns an error code.

Method calcEigenValuesAndVectors

Calculate the real eigenvalues and eigenvectors for the source matrix, A.

```
public int calcEigenValuesAndVectors( DMatBase eigval, DMatBase eigvect);
```

Parameters

eigval Returns the real eigenvalues of the source matrix A.

eigvect Returns the real eigenvectors of the source matrix A.

Return Value

Returns an error code

Method checkEigenVectors

The eigenvalues and eigenvectors of a matrix must satisfy the following equation: Ax = Lx, where A is the original matrix, L is an eigenvalue of A, and x is the eigenvector of A corresponding to eigenvalue L. Given the original matrix A, and the eigenvalues and eigenvectors of A, this method calculates Ax - Lx for each eigenvalue/eigenvector pair and compares the result to a threshold residual, returning true if the actual residual is less than the test threshold residual. This routine assumes the matrices use base-zero [0..N-1] indices.

```
public static boolean checkEigenVectors(
   DMatBase A,
   DMatBase evalues,
   DMatBase evectors,
   DMatBase residuals,
   double [] resid
);


public static boolean checkEigenVectors(
   DMatBase A,
   XMatBase evalues,
   XMatBase evectors,
   XMatBase residuals,
   double [] resid
);
```

Parameters

A The original real matrix.

evalues A vector of the eigenvalues for the matrix A.

evectors A matrix of the eigenvectors for the matrix A.

residuals Returns a vector containing the actual residuals of Ax - Lx.

resid A reference number, if all residuals of ABS(Ax - Lx) are less than resid, the method returns true. The maximum error is returned in *resid[0]*.

Return Value

Returns true if the all of the residuals of Ax - Lx are less than the *resid[0]* value.

Example

Code listing extracted from the EigenRealSym example program.

```
DDMat A = new DDMat();
.
. // Matrix A needs to be initialized
.
// results returned in EValues and EVectors
DDMat EValues = new DDMat();
DDMat EVectors = new DDMat();
.
.
EigenQL eigensolve = new EigenQL(A);
eigensolve.calcEigenValuesAndVectors(EValues, EVectors);

DDMat residuals = new DDMat(); // residuals returned here
double  [] refnum = {1.0e-4}; // if any residual > this value, solution
                        // invalid, returns actual max error
boolean ok = EigenBase.checkEigenVectors(A, EValues, EVectors, residuals, refnum);
```

QR Algorithm for Real Unsymmetric Matrices

Class EigenQR

The **EigenQR** class will calculate all of the eigenvalues a real-symmetric matrix using the QR method. If you request the eigenvalues and eigenvectors from this class, it in turn calls the **EigenQR2** class, because that is the one that calculates eigenvectors for a real-unsymmetric matrix.

EigenQR Constructor

```
public EigenQR(  DMatBase a );
```

Parameters

a The source matrix for the eigensystem calculations. The source matrix must be square, but it does not have to be symmetric.

Method calcEigenValues

Calculate the real eigenvalues for the source matrix, A. Eigenvalues that have an imaginary part are discarded. Since a real unsymmetric matrix can produce real and complex eigenvalues, you probably want to be calling the **calcEigenValues** method (next) that accepts a complex matrix instead.

```
public int calcEigenValues( DMatBase eigval);
```

Calculate the complex eigenvalues for the source matrix, A.

```
public int calcEigenValues( XMatBase eigval);
```

Parameters

eigval Returns the eigenvalues of the source matrix A.

Return Value

Returns an error code.

Method calcEigenValuesAndVectors

Calculate the real eigenvalues and eigenvectors for the source matrix, A. Eigenvalues that have an imaginary part are discarded. Since a real unsymmetric matrix can produce real and complex eigenvalues, you probably want to be calling the **CalcEigenValues** method (next) that accepts a complex matrix instead.

```
public int calcEigenValuesAndVectors( DMatBase eigval, DMatBase eigvect);
```

Calculate the complex eigenvalues and eigenvectors for the source matrix, A.

```
public int calcEigenValuesAndVectors( XMatBase eigval, XMatBase eigvect);
```

Parameters

eigval Returns the eigenvalues of the source matrix A.

eigvect Returns the eigenvectors of the source matrix A.

Return Value

Returns an error code

Example

See code listing under EigenQR2.

Class EigenQR2

The **EigenQR** class will calculate all of the eigenvalues and eigenvectors for a real-symmetric matrix using the implicit QR method.

EigenQR2 Constructor

```
public EigenQR2(  DMatBase a );
```

Parameters

a The source matrix for the eigensystem calculations. The source matrix must be square, but it does not have to be symmetric.

Method calcEigenValues

Calculate the complex eigenvalues for the source matrix, A.

```
public int calcEigenValues( XMatBase eigval);
```

Parameters

eigval Returns the eigenvalues of the source matrix A.

Return Value

Returns an error code.

Method calcEigenValuesAndVectors

Calculate the complex eigenvalues and eigenvectors for the source matrix, A.

```
public int CalcEigenValuesAndVectors( XMatBase eigval, XMatBase eigvect);
```

Parameters

eigval Returns the eigenvalues of the source matrix A.

eigvect Returns the eigenvectors of the source matrix A.

Return Value

Returns an error code

Example

Code listing extracted from the EigenRealUnSym example program.

```
DDMat A = new DDMat();
.
. // Matrix A needs to be initialized
.
// results returned in EValues and EVectors
DXMat EValues = new DXMat();
DXMat EVectors = new DXMat();.
.
.
EigenQR2 eigensolve = new EigenQR2(A);
eigensolve.calcEigenValuesAndVectors(EValues, EVectors);

DXMat residuals = new DXMat(); // residuals returned here
double [] refnum = {1.0e-4}; // if any residual > this value,
                        // solution invalid, returns actual max error
boolean ok = EigenXBase.checkEigenVectors(A, EValues, EVectors, residuals, ref
refnum);
```

LR Algorithm for Complex Matrices

Class EigenXLR

The **EigenXLR** class will calculate all of the eigenvalues and eigenvectors for a complex matrix using the LR method. There are no symmetry constraints.

EigenXLR Constructor

```
public EigenXLR(  XMatBase a );
```

Parameters

a The source matrix for the eigensystem calculations. The source matrix must be square, but it does not have to be symmetric.

Method calcEigenValues

Calculate the complex eigenvalues for the source matrix, A.

```
public int calcEigenValues( XMatBase eigval);
```

Parameters

eigval Returns the eigenvalues of the source matrix A.

Return Value

Returns an error code.

Method calcEigenValuesAndVectors

Calculate the complex eigenvalues and eigenvectors for the source matrix, A.

```
public int calcEigenValuesAndVectors( XMatBase eigval, XMatBase eigvect);
```

Parameters

eigval Returns the eigenvalues of the source matrix A.

eigvect Returns the eigenvectors of the source matrix A.

Return Value

Returns an error code

Method EigenXBase.checkEigenVectors

The eigenvalues and eigenvectors of a matrix must satisfy the following equation: Ax = Lx, where A is the original matrix, L is an eigenvalue of A, and x is the eigenvector of A corresponding to eigenvalue L. Given the original matrix A, and the eigenvalues and eigenvectors of A, this method calculates Ax - Lx for each eigenvalue/eigenvector pair and compares the result to a threshold residual, returning true if the actual residual is less than the test threshold residual. This routine assumes the matrices use base-zero [0..N-1] indices.

```
public static boolean checkEigenVectors(
   XMatBase A,
   XMatBase evalues,
   XMatBase evectors,
   XMatBase residuals,
   double [] resid
);
```

Parameters

A The original, complex, matrix.

evalues A vector of the eigenvalues for the matrix A.

evectors A matrix of the eigenvectors for the matrix A.

residuals Returns a vector containing the actual residuals of Ax - Lx.

resid A reference number, if all residuals of ABS(Ax - Lx) is less than *resid[0]*, the method returns true.

Return Value

Returns true if the all of the residuals of Ax - Lx are less than the *resid[0]* value.

Example

Code listing extracted from the EigenComplex example program

```
DXMat A= new DXMat();
.
. // Initialize complex matrix
.
DXMat EValues1 = new DXMat();
DXMat EVectors1 = new DXMat();

EigenXLR eigensolve1 = new EigenXLR(A);
eigensolve1.calcEigenValuesAndVectors(EValues1, EVectors1);

DXMat residuals = new DXMat();
double [] resid = {1.0e-5};
boolean ok1 = EigenXLR.checkEigenVectors(A, EValues1,
```

```
                EVectors1, residuals, ref  resid);


DXMat residuals = new DXMat(); // residuals returned here
Double [] refnum = {1.0e-4}; // if any residual > this value, solution invalid,
returns
                        // actual max error
boolean ok = EigenXBase.checkEigenVectors(A, EValues, EVectors, residuals,
refnum);
```

Chapter 7 – Multiple Regression

Regression analysis is a statistical tool for the evaluation of the relationship of one or more independent variables $X_1, X_2, \ldots X_k$ to a single dependent variable Y for a fixed number of observations. The primary goal of regression analysis is to obtain predictions of Y using the known values of X. These predictions are made using a regression equation. They are subject to error and are only estimates of the true values of Y. There can be other goals of regression analysis, for example, to determine which of several independent variables are important and which are not for describing and predicting a dependent variable. The simplest case of regression is known as simple linear regression and it calculates the relationship between one variable x and another one y. The relationship is considered linear because it is assumed that the underlying equation relating the two variables is of the form: y = mx + b.

Simple linear regression would take a list of (x,y) pairs (2 would be the minimum number of pairs) and calculate the slope coefficient, m, and the offset coefficient, *b*. Regression does not always return the exact values for the slope and offset because the data is usually empirical. The calculated slope and offset represent the best trend line which can be fitted through the data. The estimate is calculated using an algorithm which minimizes the sum of the squares of the vertical deviations separating the trend line from the actual data points. This algorithm is known as the method of least squares and it is by far the most common technique used in regression analysis.

Simple linear regression only handles equations for two variables. Multiple linear regression deals with equations of at least two variables. It is quite possible to provide estimates of equations of 100 or more variables using multiple regression. Large economic forecasting models use multiple regression to analyze data involving hundreds of variables.

The general form of a regression equation or model for *k* independent variables is given by the equation:

$$Y = b_0 + b_1 X_1 + b_2 X_2 + \ldots + b_k X_k$$

where $b_0, b_1, b_2, \ldots b_k$ are the regression coefficients that need to be estimated. The variable Y is the dependent variable in the equation above. The variables, $X_1, X_2, \ldots X_k$, are the independent variables. The objective of regression is to calculate the coefficients $b_0, b_1, b_2, \ldots b_k$ in order to produce the best estimates of the dependent variable Y given the values of the independent variables $X_1, X_2, \ldots X_k$.

The independent and dependent variables are all measured values. To calculate the coefficients you must first acquire enough data so that you have at least one more set of observations than the number of variables you are trying to fit. One of the most important measures of the quality of the regression equation fit is known as the correlation coefficient. The correlation coefficient is usually abbreviated as the r-value (R) of the

regression. The closer this value is to 1, the better the fit. Another measure of the fit is called the coefficient of determination, usually referred to as the r^2 (R^2) value. This value is equal to the square of the correlation coefficient. Again, the closer this value is to 1, the better the fit. The standard error of the estimate (usually abbreviated SEE), is yet another measure of the quality of the regression fit. It is a measure of the scatter of the actual data about the regression line. The smaller the SEE value, the closer the actual data is to the theoretical regression line. If you would like to learn more about the uses of regression, any good college textbook on statistics should cover the basics. Many graduate school textbooks are devoted entirely to the uses and interpretation of the results of regression. The mathematics behind the actual calculations involved in least squares multiple regression can be found in textbooks devoted to linear algebra and matrix math.

We provide several algorithms multiple regression – the standard least squares approach, which regresses only the independent variables you specify, and three other methods, forward selection, backward elimination and stepwise selection, which automatically choose the final regression coefficients, starting with your best guess, and eliminating variables with marginal F-Test statistics. They have different properties and each one can be the best for a particular regression problem and specific data set. As a rule, in case of not very highly correlated independent variables, the model obtained using general standard least squares regression will produce the smallest error if checked on the same data set on which it was built. The other methods produce results slightly less accurate, but simpler and more reliable, models.

Least Squares Multiple Regression

Class MulReg

MulReg

The **MulReg** class uses the least squares method of fitting data to an equation. It chooses regression equation coefficients that minimize the sum of the squares of the distances between the observed responses and those predicted by the fitted model. This sum of squares is called the residual sum of squares, or SEE. The class includes in the final regression equation all of the independent variables, except for those explicitly excluded by the consideration property or method.

In matrix math form, the algorithm is as follows:

X = matrix of independent variable values,

Y = vector of dependent variable values,

B = vector of regression coefficients to solve for.

The objective of the regression is to solve for B given X and Y. First, the data matrix X is converted into the square matrix A by multiplying it by its own transpose.

$$A = X^T X$$

A constant vector C is calculated by multiplying the transposed data matrix X by the dependent variable Y.

$$C = X^T Y$$

The solution vector B is calculated by solving the linear system of equations (remember Ax = b)

$$AB = C,$$

for the unknown B, using one of the equation solving algorithms: Gauss-Jordan, LU decomposition, Cholesky decomposition, or QR factorization. The Cholesky decomposition algorithm can be used because the A matrix ($X^T X$) will always be symmetric, positive, definite. B is a vector that returns the coefficients of the least squares multiple regression fit to the original data.

This method works well if the independent variables are not highly correlated with each other, and if most of them contribute significantly to the dependent variable. Otherwise, it may encounter problems. For example, if two of the independent variables are highly, or perfectly, correlated, the linear equations may become ill-conditioned and produce erroneous results. If there are many independent variables which are not correlated with the dependent one, they will never less all be included in the model and the model may be unreliable as a result.

MulReg constructor

Set up a multiple regression, including all of the independent variables in the final solution.

```
public MulReg(
   DMatBase iv,
   DMatBase dv
);
```

Set up a multiple regression, including only the independent variables flagged by the *consid* vector in the final solution.

```
public MulReg(
   DMatBase iv,
   DMatBase dv,
   IVect consid
);
```

iv A matrix of the independent variable (x-values).
dv A matrix of the dependent variable (y-values).
consid A vector contains consideration values, one for each independent variable i the *iv* array, plus the constant. Element 0 contains the constant flag, the other elements contain the flags for the iv array. Initialize the array elements using one of the consider constants: REG_VARIN, REG_VAROUT, REG_VARCONSIDER. If an element is REG_VARIN, or REG_VARCONSIDER, the corresponding variable is included in the regression.

Use the **solve** method to solve the current multiple regression using the current parameters. The methods returns the regression equation coefficients, and summary regression statistics.

Method MulReg.solve

```
public int solve(
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

regcoef Returns the calculated regression coefficients.
regstats Returns regression statistics. The **RegStats** class is a simple class that encapsulates several related summary statistics for multiple regression:

RegStats class

Public Instance Properties

ovfst	Get/Set overall F statistic
r	Get/Set multiple correlation coefficient (R-value)
rsq	Get/Set coefficient of determination (R-Squared)
see	Get/Set Returns the Standard error of estimate (
sigflag	Get/Set significance of regression flag
sumressq	Get/Set sum of residuals squared

Multiple regression example – Extracted from the example program Mulregtest.

```
MulReg mulreg = new MulReg();
DDMat A = new DDMat(); // Independent variable matrix
DDMat RHS = new DDMat(); // Ddependent variable matrix
.
.
// Initialize A and RHS matrices;

mulreg = new MulReg (A, RHS);
for (int i=0; i < mulreg.getIndependentVariables().numColumns + 1; i++)
    mulreg.setConsider(i,MulReg.REG_VARIN);
RegStats regstats = new RegStats();
DDMat coefficients = new DDMat();
mulreg.solve(coefficients, regstats);
```

Other useful regression statistics can be calculated on demand. These include the partial F- statistics. F-Test statistics can take a lot of time to calculated, since it involves as many separate multiple regressions as there are independent variables. Call the method **FTestAnalysis** to calculate the F-statistics.

Method MulReg.FTestAnalysis

```
public int FTestAnalysis(
   DDMat fstat,
   double conf_level,
   DDMat coefsig
);
```

Parameters

fstat Returns the calculated f-statistics, one for each independent variable.

conf_level A fractional value (usually between 0.8 to 0.99) specifying the confidence level. An independent variable is considered significant if the probability of it being significant exceeds this confidence level.

coefsig Returns an array of flags, one for each independent variable. If the value of an array element is 1, that means that the probably of the corresponding independent variable being significant exceeds the specified confidence level.

Automatic Selection of Regression Variables

Class StepwiseMulReg

The **MulReg** class allows the programmer to include and exclude independent variables from the final regression equation using the consider (*consid*) parameter of the **MulReg** constructor. Another approach is to let the program decide what independent variables to include in the final regression equation. The **StepwiseMulReg** class includes three algorithms for the automatic selection of the independent variables used in the regression model: backward elimination, forward selection, and stepwise selection. You can still force variables in and out using the *consider* parameter of the constructor. The selection mechanism will choose between variables marked REG_VARCONSIDER. Those marked with REG_VARIN will be included in the model, no matter how insignificant they are found to be. Those marked with REG_VAROUT will be excluded from of the model.

Setup up the regression the problem using the **StepwiseMulReg** constructor.

StepwiseMulReg constructor

Set up a stepwise multiple regression, including all of the independent variables in the final solution.

```
public StepwiseMulReg(
   DMatBase iv,
   DMatBase dv
);
```

Set up a stepwise multiple regression, including only the independent variables flagged by the *consid* vector in the final solution.

```
public StepwiseMulReg(
   DMatBase iv,
   DMatBase dv,
   IVect consid
);
```

iv A matrix of the independent variable (x-values).
dv A matrix of the dependent variable (y-values).
consid A vector contains consideration values, one for each independent variable i the iv array, plus the constant. Element 0 contains the constant flag, the other elements contain the flags for the iv array. Initialize the array elements using one of the consider constants: REG_VARIN, REG_VAROUT, REG_VARCONSIDER. If an element is REG_VARIN it is always included in the final regression equation. If an element is REG_VAROUT it is excluded from the final regression equation. If an element is REG_VARCONSIDER, the corresponding variable available for selection into the final regression equation.

Backward Elimination

The backward elimination algorithm starts with a model that includes all of the variables marked with REG_VARIN and REG_VARCONSIDER. The basic algorithm is as follows:

Step 1. Solve for the current regression equation.

Step 2. Calculate the partial F statistic for every variable in the model.

Step 3. Find the variable with the smallest partial F statistic and using that value, compare the probability that the variable is significant to the value of the property **FToLeave**. If the probability that the variable is significant is less than **FToLeave**, the variable is removed from the model.

Step 4. Repeat, starting at Step 1, until the significance test in Step 3 finds no variable to remove. The current solution then becomes the final solution of the regression equation.

Method StepwiseMulReg.backwardSolve

```
public int BackwardSolve(
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

regcoef Returns the calculated regression coefficients.

regstats Returns regression statistics. The **RegStats** class is a simple class that encapsulates several related summary statistics for multiple regression. See the description under the **MulReg** class section.

Forward Selection

The forward selection algorithm starts with a model that includes only the variables marked with REG_VARIN. If there are no variables marked with REG_VARIN, then the model starts with the variable that has the highest correlation with the dependent variable.

Step 1. Select the starting variable(s). This is either the variables marked with REG_VARIN, or the variable with the highest correlation with the dependent variable.

Step 2. Calculate the partial F statistic for every other variable marked with REG_VARCONSIDER.

Step 3. Find the variable with the largest partial F statistic and using that value, compare the probability that the variable is significant to the value of the property FToEnter. If the probability that the variable is significant is greater than FToEnter, the variable is added to the model.

Step 4. Repeat, starting at Step 1, until the significance test in Step 3 finds no additional variables to add. The current solution then becomes the final solution of the regression equation.

Method StepwiseMulReg.forwardSolve

```
public int ForwardSolve  (
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

regcoef Returns the calculated regression coefficients.

regstats Returns regression statistics. The RegStats class is a simple class that encapsulates several related summary statistics for multiple regression. See the description under the MulReg class section

Stepwise Selection

The stepwise selection algorithm is similar to the forward selection algorithm, with an added step. After a variable is found significant and added to the model, the model is checked again to see if any of the partial F statistics fall below a predetermined minimum. In this case the variable is removed from the model. Variables might be removed in this fashion because of auto-correlation between other variables in the model.

Step 1. Select the starting variable(s). This is either the variables marked with REG_VARIN, or the variable with the highest correlation with the dependent variable.

Step 2. Calculate the partial F statistic for every other variable marked with REG_VARCONSIDER.

Step 3. Find the variable with the largest partial F statistic and using that value, compare the probability that the variable is significant to the value of the property FToEnter. If the probability that the variable is significant is greater than FToEnter, the variable is added to the model.

Step 4. Recalculate the model and find the variable with the smallest partial F statistic and using that value, compare the probability that the variable is significant to the value of the property FToLeave. If the probability that the variable is significant is less than FToLeave, the variable is removed from the model.

Step 5. Repeat, starting at Step 1, until the significance test in Step 3 finds no additional variables to add. The current solution then becomes the final solution of the regression equation.

Method StepwiseMulReg.stepwiseSolve

```
public int StepwiseSolve  (
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

regcoef Returns the calculated regression coefficients.

regstats Returns regression statistics. The **RegStats** class is a simple class that encapsulates several related summary statistics for multiple regression. See the description under the **MulReg** class section

Examples for Stepwise Regression

The follow code fragments are extracted from the StepRegTest example program.

```
[C#]
StepwiseMulReg mulreg = new StepwiseMulReg ();
DDMat A = new DDMat(); // Independent variable matrix
DDMat RHS = new DDMat(); // Ddependent variable matrix
.
.
// Initialize A and RHS matrices;

mulreg = new StepwiseMulReg (A, RHS);
for (int i=0; i < mulreg.getIndependentVariables().numColumns + 1; i++)
    mulreg.setConsider(i,StepwiseMulReg.REG_VARCONSIDER);

RegStats regstats = new RegStats();
DDMat coefficients = new DDMat();
regmode = GetRegMode();
switch (regmode)
{
    case 0:  mulreg.solve(coefficients, regstats ); break;
    case 1: mulreg.forwardSolve(coefficients, regstats );  break;
    case 2: mulreg.backwardSolve(coefficients, regstats );  break;
    case 3: mulreg.stepwiseSolve(coefficients, regstats ); break;
   default:  mulreg.solve(coefficients, regstats ); break;
}
```

Class SplitMulReg

What is the goal of your regression? If you want to find the best model that gives the smallest error for the existing data you should be using one of the stepwise multiple regression algorithms. If you want to create a model that is the best predictor for new sample data, then you should try the **SplitMulReg** algorithm. This means that you are looking for a model that gives the best prediction of Y, given $X_1, X_2, \ldots X_k$ for some new observations.

The **SplitMulReg** algorithm is a modification of a stepwise regression that randomly assigns the raw sample data into two groups, a training group, and the holdout group. The original independent variables are replaced with their orthogonalized equivalents, so that inclusion of a new variable does not affect the coefficients for those already included in the model. At every step the independent variable with the smallest difference of coefficients determined independently on both data groups is chosen.

The algorithm does not involve the solution of a system of linear and is not sensitive to the colinearity of the independent variables. Depending on the data, the residual error on the original data may be higher than that using one of the other multiple regression algorithms, but the regression equation is usually more simple and reliable.

The **StepMulReg** class allows the programmer to include and exclude independent variables from the final regression equation using the consider (*consid*) parameter of the **StepMulReg** constructor. The selection mechanism will choose between variables marked REG_VARCONSIDER. Those marked with REG_VARIN will be included in the model, no matter how insignificant they are found to be. Those marked with REG_VAROUT will be excluded from of the model.

Setup up the regression the problem using the **SplitMulReg** constructor.

SplitMulReg constructor

Set up a split multiple regression, including all of the independent variables in the final solution.

```
public SplitMulReg (
   DMatBase iv,
   DMatBase dv
);
```

Set up a split multiple regression, including only the independent variables flagged by the *consid* vector in the final solution.

```
public SplitMulReg (
   DMatBase iv,
   DMatBase dv,
   IVect consid
);
```

iv A matrix of the independent variable (x-values).
dv A matrix of the dependent variable (y-values).
consid A vector contains consideration values, one for each independent variable i the iv array, plus the constant. Element 0 contains the constant flag, the other elements contain the flags for the iv array. Initialize the array elements using one of the consider constants: REG_VARIN, REG_VAROUT, REG_VARCONSIDER. If an element is REG_VARIN it is always included in the final regression equation. If an element is REG_VAROUT it is excluded from the final regression equation. If an element is REG_VARCONSIDER, the corresponding variable available for selection into the final regression equation.

Use the **Solve** method to solve the current multiple regression using the current parameters. The methods returns the regression equation coefficients, and summary regression statistics.

Method SplitMulReg.solve

```
public int solve(
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

regcoef Returns the calculated regression coefficients.
regstats Returns regression statistics. The **RegStats** class is a simple class that encapsulates several related summary statistics for multiple regression:

8. Curve Fitting

CurveFit
CubicSplines

Experimental data often takes the form of measurements of some kind (temperature or volts for example) taken at discrete and possibly periodic times. In such cases a table of the measured variable versus time is produced. The goal of the research is to uncover the underlying function f(x), measured for the interval $[x_0..x_n]$. Once the underlying function is discovered (the solution is usually only an approximation of the true function), the function can be used to calculate the values of f(x) (temperature in this case) for x values (points in time) which were not part of the original measured data. Polynomials are generally used as approximating functions because of their simplicity and speed of calculation on computers. There are several ways to use polynomials to approximate a discrete function. The first is to fit a single polynomial to a group of data points. This is called *polynomial curve fitting* and that is what our **CurveFit** class does. The second method starts with the assumption that the curve between each contiguous pair of data points is best modeled using a unique cubic (polynomial of degree 3) equation. This is called *cubic splines* and that is what our **CubicSplines** class does.

Polynomial Curve Fitting

Polynomial curve fitting results in a single polynomial equation of order *n* which is the least squares approximation of the original data.

Eqn. 1 $$Y = C_0 + C_1 * X^1 + C_2 * X^2 + C_3 * X^3 + ... C_n * X^n$$

Predict what the value of Y will be for a given X by plugging a value for X into the equation and solving it for Y. One of the most common applications of polynomial curve fitting is thermocouple linearization where the voltage present at a thermocouple junction is transformed into temperature by applying a polynomial equation which has been solved for using these types of curve fitting techniques.

Polynomial curve fitting is actually a special case of least squares multiple regression. Eqn. 1 above is transformed into the standard multiple regression equation:

Eqn. 2 $$Y = C_0 + C_1 * X_1 + C_2 * X_2 + C_3 * X_3 + ... C_n * X_n$$

using the transforms:

$X_1 = X$

$X_2 = X^2$

$X_3 = X^3$

.

.

.

$X_n = X^n$

A system with one dependent variable Y and one independent variable X is turned into a system of one dependent variable Y and as many independent variables (X_1, X_2 ...X_n) as the order of the curve fit. Each new independent variable is the original independent variable raised to some power. Once the transform is carried out, the multiple regression function is called to calculate the solution coefficients C_0 through C_n. One of the most important measures of the quality of the polynomial fit is known as the correlation coefficient. The correlation coefficient is usually abbreviated as the r-value (R) of the fit. The closer this value is to 1, the better the fit. Another measure of the fit is called the coefficient of determination, usually referred to as the r-squared (R^2) value. This value is equal to the square of the correlation coefficient. Again, the closer this value is to 1, the better the fit. The standard error of the estimate (usually abbreviated SEE), is yet another measure of the quality of the fit. It is a measure of the scatter of the actual data about the fitted line. The smaller the SEE value, the closer the actual data is to the theoretical polynomial equation. Other equations, such as rational functions, exponentials, etc. also can be used for curve fitting. These functions are handled in a manner analogous to polynomial curve fitting. The original data is transformed in a manner which reduces a non-linear curve fitting problem into linear curve fitting problem which can be solved using linear multiple regression.

Class CurveFit

The **CurveFit** class will fit a polynomial equation to a set of data representing discrete samples of some function y = f(x). It can also fit exponential, rational polynomial and mixed equation to the same data. A polynomial curve fit is represented by the equation:

$y = f(x) \rightarrow \quad y = a_0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + ... a_n * x^n$

The polynomial curve fit algorithm solves for the coefficients a_0 to a_n.

An exponential curve fit is represented by the equation:

$y = f(x) \rightarrow y = ae^{bx}$

The exponential curve fit algorithm solves for the coefficients a and b. The rational polynomial is represented by the equation:

$$y = f(x) \rightarrow y = \frac{a_0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + ... a_n * x^n}{1 + b_1 * x^1 + b_2 * x^2 + b_3 * x^3 + ... b_n * x^n}$$

The rational polynomial curve fit algorithm solves for the coefficients a_0 to a_n and b_1 to b_n.

The mixed fit curve fit algorithm solves for a linear combination of the polynomial, exponential, and rational polynomial curve fits, represented using the equation:

$$y = f(x) \rightarrow y = a_0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + ... a_n * x^n + b_1 / x^1 + b_2 / x^2 \ ... \ b_n / x^n + c * \ln(x).$$

CurveFit constructor

```
public CurveFit(
   DMatBase iv,
   DMatBase dv
);
```

iv A vector of the independent variable (x-values).
dv A vector of the dependent variable (y-values).

The four curve fit algorithms all use the constructor above. Call the solve method appropriate to the algorithm you want to use.

Method CurveFit.polynomialSolve

This method fits the data to a polynomial equation of the form:

$$y = a_0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + ... a_n * x^n$$

where *n* is considered the order of the polynomial. The p**olynomialSolve** method returns the solution coefficients of the polynomial (a_0 … a_n) summary statistics for the curve fit.

```
public int polynomialSolve(
   int order,
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

order Set the maximum order of the polynomial.
regcoef Returns the calculated curve fit coefficients. The coefficients are returned in an array, elements [0] to [order].
regstats Returns curve fit statistics.

Return Value

Returns an error code.

RegStats class

Public Instance Properties

ovfst	Get/Set overall F statistic
r	Get/Set multiple correlation coefficient (R-value)
rsq	Get/Set coefficient of determination (R-Squared)
see	Get/Set Returns the Standard error of estimate (
sigflag	Get/Set significance of regression flag
sumressq	Get/Set sum of residuals squared

Method CurveFit.exponentialSolve

This method fits the data to an exponential equation of the form:

$$y = ae^{bx}.$$

The **exponentialSolve** method returns the solution coefficients *a* and *b* in the equation above. All values of y must be positive.

```
public int exponentialSolve (
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

regcoef Returns the calculated curve fit coefficients. The (a) coefficient is returned in the array element [0] and the (b) coefficient is returned in the array element [1].

regstats Returns curve fit statistics. The **RegStats** class is described under the **CurveFit.polynomialSolve** section.

Return Value

Returns an error code.

Method CurveFit.rationalSolve

This method fits the data to a rational polynomial equation of the form:

$$y = \frac{a_0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + ... a_n * x^n}{1 + b_1 * x^1 + b_2 * x^2 + b_3 * x^3 + ... b_n * x^n}$$

The r**ationalSolve** method returns the solution coefficients a_0 … a_n and $b_1 + b_n$ in the equation above.

```
public int rationalSolve(
   int order,
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

order Set the maximum order of the polynomial.

regcoef Returns the calculated curve fit coefficients. The (a) coefficients are returned in array element [0..order] and the (b) coefficients are returned in array elements [order+1..2 * order].

regstats Returns curve fit statistics. The **RegStats** class is described under the **CurveFit.polynomialSolve** section.

Return Value

Returns an error code.

Method CurveFit.mixedSolve

This method fits the data to a mixed polynomial equation of the form:

$$y = a_0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + ... a_n * x^n +$$

$$b_1 / x^1 + b_2 / x^2 \ ... \ b_n / x^n +$$

c * ln(x).

The **mixedSolve** method returns the solution coefficients $a_0 \ldots a_n$, $b_1 + b_n$, and c in the equation above.

```
public int mixedSolve(
   int order,
   DDMat regcoef,
   RegStats regstats
);
```

Parameters

order	Set the maximum order of the polynomials.
regcoef	Returns the calculated curve fit coefficients. The (a) coefficients are returned in array element [0..order], the (b) coefficients are returned in array elements [order+1..2 * order], and c is returned in element [2 * order + 1].
regstats	Returns curve fit statistics. The **RegStats** class is described under the **CurveFit.polynomialSolve** section.

Return Value

Returns an error code.

Curve fitting example – Extracted from the example program CurveFitTest.

```
CurveFit curvefit = null;
int numObs = 100;
int simOrder  = 6;
int solveOrder  = 3;
int cftype = 0;
double simErr = 0.1;
DDMat A = new DDMat(); // Independent variable matrix
DDMat RHS = new DDMat(); // Dependent variable matrix
DDMat Coefficients = new DDMat();
DDMat fstats = null;
DDMat coefdev = null;
RegStats regstats = new RegStats();
.
.  // Initialize data
.
curvefit = new CurveFit(A, RHS);

cftype = getCurveFitAlgorithm();
```

```
switch (cftype)
{
    case 1: curvefit.polynomialSolve(solveOrder, Coefficients, regstats); break;
    case 2: curvefit.exponentialSolve( Coefficients, regstats); break;
    case 3: curvefit.rationalSolve(solveOrder, Coefficients, regstats); break;
    case 4: curvefit.mixedSolve(solveOrder, Coefficients, regstats); break;
    default: curvefit.polynomialSolve(solveOrder, Coefficients, regstats); break;
}

fstats = curvefit.getFStats();
coefdev = curvefit.getCoefficientDev();
```

Cubic Splines

The cubic splines algorithm starts with the assumption that the curve between each contiguous pair of data points

(x_0, y_0)-(x_1,y_1)

(x_1, y_1)-(x_2,y_2)

(x_2, y_2)-(x_3,y_3)

.

.

(x_{n-1}, y_{n-1})-(x_n,y_n)

is best modeled using a unique cubic (polynomial of degree 3) equation. If you start with (0..n) data points (for a total of n+1 data points), you end up with *n* cubic equations, one for each segment between the data points. These cubic equations are used interpolate an estimate for f(x), for any x, in the interval $[x_0..x_n]$. Additional constraints force continuity between the first and second derivatives for each curve as you pass from one segment to the next, within the interval $[x_0..x_n]$. The continuity of the derivatives is what gives the resulting curve a smooth look analogous to the wooden *spline* that was once used by draughtmen to draw a smooth line through a series of points defining a curve.

The cubic splines algorithm implementation creates a set of equations, (n-1 x n-1), based on the constraints that the y-value, and the first and second derivatives at each of the interior nodes, must match for each interval in the range $[x_0..x_n]$. Because each segment is only linked to its nearest neighbors, the resulting system of equations is tridiagonal. A tridiagonal system of equations can be solved using a specialized algorithm that is orders

of magnitudes faster than generalized methods such as Gauss-Jordan and LU decomposition. The solution is used to created the table (n x 4) of cubic equation coefficients that are used to interpolate the f(x) values for x in the range $[x_0..x_n]$.

Cubic splines has several limitation that you must be aware of before using it.

- The raw data must be monotonic in x. This means that the data must be sorted in increasing order by x and that the x-values must always be increasing. You cannot have exactly the same x-value repeated twice.

- The data cannot reflect a function that has a discontinuous first or second derivative. A good example of a discontinuous second derivative is a step change in the function. This also violates the previous limitation, since in order to implement a true step change, a given f(x) value is associated with two x values. For example, the line segments represented by the data points:

  Point 1 (0,0)

  Point 2 (1,0)

  Point 3 (1,1)

  Point 4 (2,1)

  exhibit a discontinuity between points 2 and 3, where a step change occurs and the second derivative becomes infinite. If you are close to a step change, the cubic splines algorithm my produce while swings around in the evaluated function in the transition area.

- Cubic splines can only be used to interpolate data points in the original, fitted data interval defined by $[x_0..x_n]$. If you attempt to calculate an f(x) value outside of the interval, the algorithm will use the cubic equation of the nearest segment, either the $[x_0 ..x_1]$ or the $[x_{n-1} ..x_n]$ segment. This can result in wild swings that are not an accurate representation of the underlying function.

- The second derivative of the function is assumed to be 0 at the endpoints x_0 and x_n. This is known as a natural spline because it assumes the shape that a wooden spline would take if the ends were left free.

Class CubicSplines

The **CubicSplines** class calculates the table of cubic spline equations used in interpolating values within the range of the original data.

CubicSplines constructor

```
public cubicSplines(
   DDMat x,
   DDMat y
);
```

Parameters

x A matrix of the independent variable (x-values).
y A matrix of the dependent variable (y-values).

CubicSplines.interpolatePoint Method

Interpolate a new y-value given an x-value and the previously calculated spline coefficients.

```
public double interpolatePoint(
   double x
);
```

Parameters

x The x-value to interpolate a new y-value for.

Return Value

Returns the interpolated y-value.

Cubic splines example – Extracted from the example program CubicSplinesTest.

```
void testCubicSplines()
{
    dataY = new DDMat(sampleN);
    dataX = new DDMat(sampleN);
    for (int i=0; i < sampleN; i++)
    {
        double x = (double) i/ 2.0;
        dataX.setElement(i, x);
        dataY.setElement(i, 3.5 * Math.cos(x));
    }

    // Concatenate dataY and dataX to display in a single matrix viewer
```

```
    DDMat dataXY = new DDMat();
    DDMat.colConcat(dataX, dataY, dataXY);

    dMatViewer1 = new DMatViewer();
    dMatViewer1.setViewMatrix(dataXY, 10, 2);
    dMatViewer1.setMatName( "Data");
    dMatViewer1.setPosition(10,10,200,300);
    dMatViewer1.setDecimals(3);
    dMatViewer1.getColumnHeads().setElement(0, "X");
    dMatViewer1.getColumnHeads().setElement(1, "Y");
    this.add(dMatViewer1);
    // dMatViewer1.updateDraw();

    csplines = new CubicSplines(dataX, dataY);
}

void interpolatePoints()
{
    interpolateY = new DDMat(interpolatedN);
    interpolateX = new DDMat(interpolatedN);
    actualY = new DDMat(interpolatedN);
    residualY = new DDMat(interpolatedN);


    // This interpolates on the same iterval as the original data
    for (int i=0; i < interpolatedN; i++)
    {
        double x =  (double) i/ 20.0;
        interpolateX.setElement(i, x);
        interpolateY.setElement(i, csplines.InterpolatePoint(x));

        actualY.setElement(i, 3.5 * Math.cos(x));

        residualY.setElement(i, interpolateY.getElement(i) -
actualY.getElement(i));

    }

// Concatenate interpolateX and interpolateY to display in a single matrix viewer
    DDMat interpolateXY = new DDMat();
    DDMat.colConcat(interpolateX,interpolateY, interpolateXY);
    DDMat.colConcat(interpolateXY,actualY, interpolateXY);
    DDMat.colConcat(interpolateXY,residualY, interpolateXY);
```

```
    dMatViewer2 = new DMatViewer();
    dMatViewer2.setViewMatrix(interpolateXY, 20, 4);
    dMatViewer2.setPosition(330,10,400, 500);
    dMatViewer2.setMatName("Data");
    dMatViewer2.setDecimals( 3);
    dMatViewer2.getColumnHeads().setElement(0, "X");
    dMatViewer2.getColumnHeads().setElement(1, "Y-Interpolate");
    dMatViewer2.getColumnHeads().setElement(2, "Y-Actual");
    dMatViewer2.getColumnHeads().setElement(3, "Y-Residual");
    this.add(dMatViewer2);
    dMatViewer2.updateDraw();

}
```

9. Digital Signal Processing

DSP

FFT Basics

A large class of numerical analysis tools involves use of Fourier analysis. Applications for the Fourier transform include speech, image, radar, and general signal processing. Implementations of the Fourier transform using sampled data on computers are generally referred to as DFT (Discrete Fourier Transform). The equation below computes the DFT, (X_k), of the input signal, x_n.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

A complex summation of N complex multiplications is required for each of N samples. This adds up to N^2 complex multiplications and N^2 complex additions to compute an N point DFT. A 1024 point DFT calculated using the equation above would require 4 million floating point multiplications and 4 million floating point additions. In the early 1960's researchers (notably Cooley and Tukey) noticed patterns in the DFT calculation that, when exploited properly, could be used to reduce the number of complex multiplications to $N * \text{Log}_2 N$. The number of floating point multiplications in a 1024 point DFT is reduced by 99%, to 40,000. The entire class of Fast DFT algorithms is known as FFT (Fast Fourier Transform).

As a historical note, while Cooley and Tukey are usually given credit for the invention of FFT, based on the publication of their algorithm in 1965, it is now known that the German mathematician Carl Frederick Gauss invented a near identical version of the FFT in the early 1800's. He used it as an aid in calculating the trajectories of asteroids and planets. Between 1800 and 1965 the FFT algorithm has been "re-invented" a number of times.

FFT Harmonics

The underlying assumption of an FFT is that an arbitrary, sampled signal can be recreated as a summation of other periodic waveforms. The FFT functions will calculate the periodic waveforms that sum up to make the original signal. The FFT functions give you enough information to calculate the frequency, magnitude and phase shift of each harmonic component. The frequency of the harmonic components in the final answer depends on the sample rate used when sampling the original, raw waveform. The FFT

can only return frequencies that are simple integer fractions of the original sampling frequency (sample frequency = SF). The range of frequencies will range from 0 (DC level or average value of the signal) to SF/2 (or the Nyquist frequency for the sample rate of SF). For example, assume that 16 samples of a periodic signal are taken at a rate of 480 Hz. The FFT algorithm will calculate the degree to which the following frequencies contribute to the signal makeup.

Harmonic	**Frequency**
0	(0/16) * 480 = 0 Hz = DC level
1	(1/16) * 480 = 30 Hz
2	(2/16) * 480 = 60 Hz
3	(3/16) * 480 = 90 Hz
4	(4/16) * 480 = 120 Hz
5	(5/16) * 480 = 150 Hz
6	(6/16) * 480 = 180 Hz
7	(7/16) * 480 = 210 Hz
8	(8/16) * 480 = 240 Hz

If you were interested in the degree to which 60 Hz (AC line frequency) noise was affecting your signal, then you could concentrate on harmonic #2 which in the example above is at 60 Hz. The general equation for calculating the frequency for a given harmonic index is:

$$F = (H/N) * SF$$

where F = frequency of harmonic H,
H = harmonic index (range 0 to N/2),
N = number of sampled data points,
SF = sample frequency.

Assume that you are sampling waveform at 50KHz and you want to resolve individual frequency components down as low as 8 Hz. How many data points do you need to sample? The formula becomes:

$$8\ Hz = (1 / N) * 50KHz, \quad \text{or}$$
$$N = (1/8) * 50KHz = 6250$$

The FFT algorithms in this package require a number of data points that is a power of 2. You should round the result to the next power of 2 above the calculated value, or 8192. The first harmonic of this sampled waveform becomes (1/8196) * 50KHz, or 6.1 Hz, that is as close as you can get to the original 8 Hz requirement without changing the sample rate.

The FFT routines use the original waveform as input. This is passed to the function as an array (vector or pointer) of floating point numbers. The output of the FFT function is one

complex number (a + b *i*) for each harmonic component in the original waveform. The output is passed back as two arrays one of which holds real parts of the complex numbers and another - imaginary parts.

The input to the FFT can be an array of real or complex numbers *(* a + b *i*). Calculate an N point complex FFT, and the primary results are returned in the first N/2 elements of the real and imaginary arrays. These elements hold the harmonic values for the positive frequencies that make up the waveform. The last half of the arrays hold the harmonic values for the negative frequencies that make up the waveform. The negative frequencies are symmetrical to the positive frequencies about the folding frequency of the harmonic (N/2). The folding frequency represented by the harmonic N/2 is also the Nyquist sample frequency for the original signal. Continuing a previous example, assume that 16 points of a signal are sampled at a rate of 480 Hz. The sampled waveform is contained in the array *w*.

w = (6.0, 1.85, -2.59, 0.765, 4.0, -0.765, -5.41, -1.85, -2.0, -1.85, -5.41, -0.765, 4.0, 0.765, -2.59, 1.85)

The FFT function returns a complex number for each positive and negative harmonic in the waveform. Assume that the real parts are returned in the vector *va* and the imaginary parts in vector *vb*. The complex number representing the value for harmonic #3 would be *va*[3] + *vb*[3] i. The complex number for the highest harmonic in a 16 point FFT would be *va*[8] + *vb*[8]i.

Many questions arise regarding interpretation of the FFT results. The absolute values of a FFT are proportional to the number of data points used in the calculation. Some FFT routines normalize the results of the FFT by dividing the calculated values by N or N/2, while the majority do not. We do not normalize the results because this would make our algorithms different from the majority of popular FFT algorithms described in the literature. This can be confusing to people who have never calculated an FFT before. If you are looking for the relative magnitude of one harmonic versus the other you can ignore the normalization factor. If you need to recover the exact value of a given harmonic, normalize the results of the FFT by N/2 if you intend to use the positive harmonics, and N if you plan to use both positive and negative harmonics. The frequency, magnitude and phase shift for each harmonic index are calculated according to the formulas below. It is assumed that results of the FFT have been returned in vector's *va* (real parts) and *vb* (imaginary parts).

DSP Windowing

Windowing functions are frequently used in digital signal processing**.** One of the common applications is to reduce the undesirable effects related to spectral leakage. Spectral leakage is the result of discontinuities in the sampled waveform. Since the waveform is sampled for a finite length of time, discontinuities occur at the end points of the waveform. Windows are weighting functions applied to the raw data to reduce the spectral leakage associated with the finite observation interval**.** A window function typically multiplies the data in the sample interval by a function which has a value of one

at its center and tapers to zero (or near zero) at both end points. The most common DSP windows are:

High and moderate resolution windows

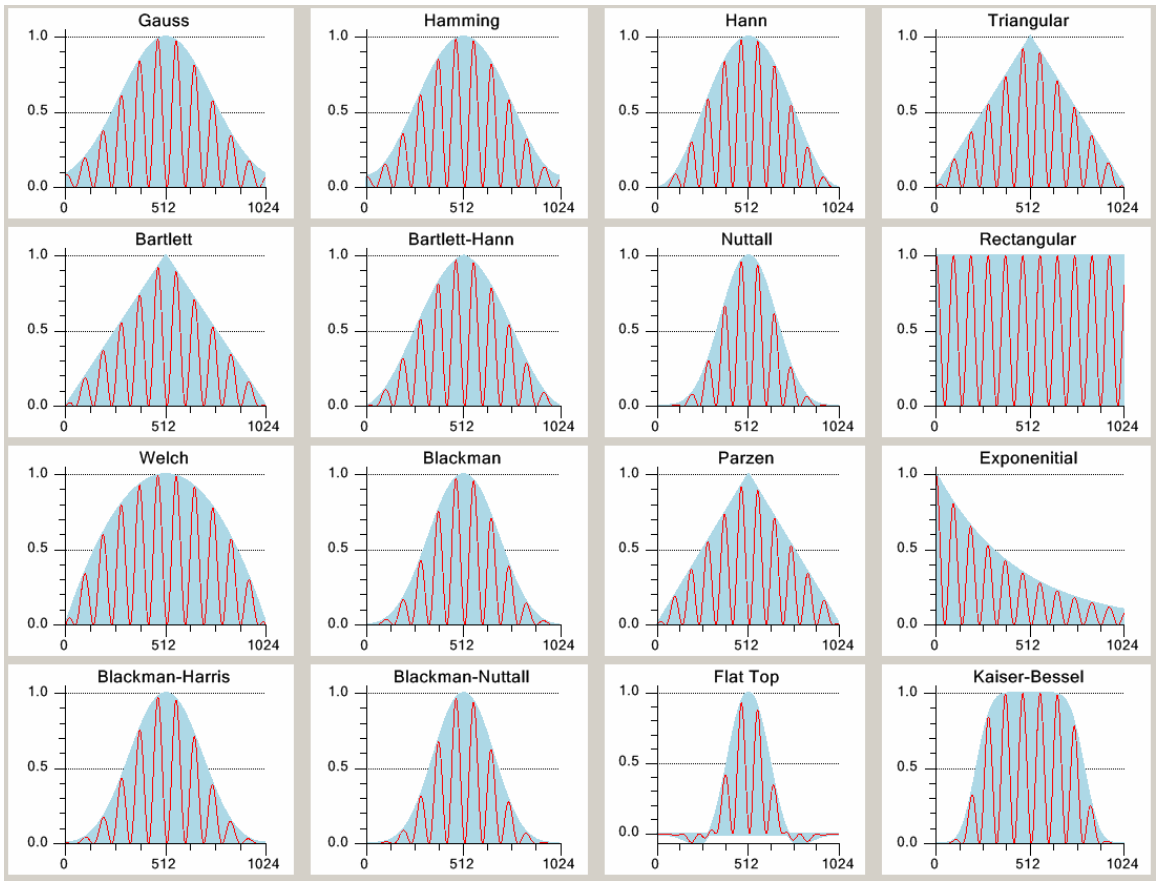
Rectangular	Gauss	Parzen	Welch
Hann (Hanning)	Hamming	Bartlett	Triangular
Bartlett-Hann	Blackman		

Low dynamic range (low resolution) windows

Nuttall	Blackman–Harris	Blackman–Nuttall	Flat top

Other

Exponential	Kaiser-Bessel



You can also apply your own custom window to the raw data. Just apply your windowing function to the raw data, and call the **FFTCalc** method that does not use a windows parameter. Or you can call the **FFTCalc** method that does specify a window parameter, and in that case specify DPS.DSPWIN.NOWIN window, since that passes through the raw data without modification.

Class DSP

The source data for an FFT can take real-only, or complex form. The raw data can be passed to the **DSP** class as simple double arrays, or as **DDMat** and **DXMat** array classes.

DSP constructor

Create a **DSP** object using double arrays.

```
public DSP(
   double[] re,
   double[] imag
);
```

re Array of real values of the raw data.
imag Array of imaginary values of the raw data. This parameter can be null for a FFT that starts with real data.

DSP constructor where the raw data is real-only and passed in using a **DDMat** matrix object.

```
public DSP(
   DDMat source
);
```

Parameters

source
Source vector of fft real data.

DSP constructor where the raw data is complex and passed in using a DXMat matrix.

```
public DSP(
   DXMat source
);
```

Parameters

source Source vector of fft complex data.

The constructors do not calculated the FFT. They just store the raw data. Calculate the actual FFT using one of the following **FFTCalc** methods.

Method DSP.fFTCalc

Calculate the forward or inverse FFT of the raw data. If the original data is real-only (no imaginary component) a real-only version of the FFT algorithm is called. If the original data is complex, a complex version of the FFT algorithm is called.

```
public DXMat fFTCalc(
   boolean bInverse
);
```

Apply a FFT window to the raw data and then calculate the forward or inverse FFT of the raw data.

```
public DXMat fFTCalc(
   boolean bInverse,
   DSPWIN fftwin
);
```

Parameters

bInverse Set to true and an inverse FFT is calculated.

fftwin DSP window type. Use one of the DSPWIN enumerated constants: NOWIN, RECTANGULAR, GAUSS, HAMMING, HANN, BARTLETT, BARTLETT_HANN, NUTTALL, BLACKMAN_HARRIS, BLACKMAN_NUTTALL, FLATTOP, PARZEN, WELCH, BLACKMAN, and TRIANGULAR, KAISER_BESSEL and EXPONENTIAL.

Return Value

Returns the FFT of the raw data as a DXMat matrix.

A few of the windows have additional parameters that you may want to change. Set the Gauss window sigma parameter using **DSP.setGaussSigma**. Set the Exponential window final value parameter using **DPS.setExpFinalValue**. Set the Kaiser-Bessel window alpha parameter using **DSP.setKBAlphaValue**.

```
// Default values
DSP.setGuassSigma(0.45);
DSP.setExpFinalValue( 0.1);
DSP.setKBAlphaValue( 1);
```

Example (Extracted from the example FFTTest)

```
int n = 4096;
double [] xR = new double[n];
double [] iR = new double[n];
for (int i=0; i < n; i++)
{
    double f = 10.0 * Math.PI * 2 * ((double) i/ (double) n);
    xR[i] = 13*Math.cos(f) + 31*Math.cos(f*2);
    // Add in a fixed offset and Randomize it a bit
    xR[i] += 4.0 + (0.5 - DMatBase.getRandomDouble());
    iR[i] = 0;
}

DDMat origData = new DDMat(xR);
DXMat FFTResult = new DXMat();

// Since iR[i] is all zeros, can use real only version of DSP
DSP fftobj = new DSP(origData);
FFTResult = fftobj.fFTCalc(false);
```

Method DSP.fFTMagnitude

Calculates the FFT magnitude associated with a given harmonic index.

```
public static double fFTMagnitude(
   XMatBase source,
   int i
);
```

Parameters

source Source vector contains the results of an FFT calculation.
i The index of the harmonic you are calculating the magnitude for.

Return Value

Returns the FFT magnitude associated with a given harmonic index.

Example

```
DDMat magdata = new DDMat(FFTResult.getLength());
```

```
for (int i=0; i < FFTResult.getLength(); i++)
        magdata.setElement(i, DSP.fFTMagnitude(FFTResult, i));
```

Calculate all of the FFT magnitudes associated with an FFT results matrix.

```
public static DDMat fFTMagnitude(
   XMatBase source
);
```

Parameters

source Source vector contains the results of an FFT calculation.

Return Value

Returns a **DDMat** matrix of the FFT magnitudes associated with the FFT data.

Example (Extracted from the example FFTTest)

```
DDMat magdata = DSP.fFTMagnitude(FFTResult);
```

Method DSP.fFTPhase

Calculates the FFT phase associated with a given harmonic index.

```
public static double fFTPhase(
   XMatBase source,
   int i
);
```

Parameters

source Source vector contains the results of an FFT calculation.
i The index of the harmonic you are calculating the phase for.

Return Value

Returns the FFT phase associated with a given harmonic index

Example

```
DDMat phasedata = new DDMat(FFTResult.getLength());
for (int i=0; i < FFTResult.Length; i++)
        pasedata.setElement(I, DSP.fFTPhase(FFTResult, i));
```

Method DSP.fFTPhase2

Calculates the FFT phases associated with already calculated fft data.

```
public static DMatBase fFTPhase2(
   XMatBase source
);
```

Parameters

source Source vector contains the results of an FFT calculation.

Return Value

Returns a **DDMat** matrix of the fft phases associated with the fft data.

Example (Extracted from the example FFTTest)

```
[C#]
DDMat phasedata = DSP.fFTPhase2(FFTResult);


[VB]
Dim phasedata As DDMat = DSP.fFTPhase2(FFTResult)
```

Method DSP.fFTFrequency2

Calculates the FFT frequencies associated with already calculated FFT data.

```
public static DDMat fFTFrequency2(
   int lNumdat,
   double rSampleFreq
);
```

Parameters

lNumdat The number of data points, must be a power of 2.
rSampleFreq The sample frequency of the raw data.

Return Value

Returns a matrix of the FFT frequencies associated with the FFT data.

Example (Extracted from the example FFTTest)

```
DDMat freqdata = DSP.fFTFrequency2(FFTResult.Length, 1000.0);
```

Power Spectrum Calculations

This function calculates the power spectrum periodogram of a sampled data set. The power spectral density of each frequency bin is returned, along with the exact frequency in Hz for each frequency bin.

Method: If N data points of a waveform $w(t)$ are collected at equal intervals, then the FFT of the waveform can be denoted as $W(t)$. The periodogram estimate of the power spectrum, $P(k)$ is defined as:

$$P_0 = \frac{1}{N^2}\left|W_0\right|^2$$

$$P_0 = \frac{1}{N^2}\left[\left|W_k\right|^2 + \left|W_{N-k}\right|^2\right]_{k = 1, 2, \ldots, (N/2 - 1)}$$

$$P_{\frac{N}{2}} = \frac{1}{N^2}\left|W_{\frac{N}{2}}\right|^2$$

where k is defined for zero and positive frequencies.

Method DSP.powerSpectrumCalc

Calculates the power spectrum periodogram of a sampled dataset.

```
public static DDMat powerSpectrum(
   XMatBase source,
   double rInterval,
   DDMat freqvalues
```

Real-only version of the power spectrum periodogram of a sampled dataset

```
public static DDMat powerSpectrum(
   DDMat source,
   double rInterval,
   DDMat freqvalues
);
```

Parameters

source	Raw data. The power spectrum routine takes the FFT of the data, pass in raw data.
rInterval	The sample period between adjacent data points in the seconds.
freqvalues	Returns the frequency values associated with each harmonic index.

Return Value

Returns the power spectrum of the original signal.

Example (Extracted from the example FFTTest)

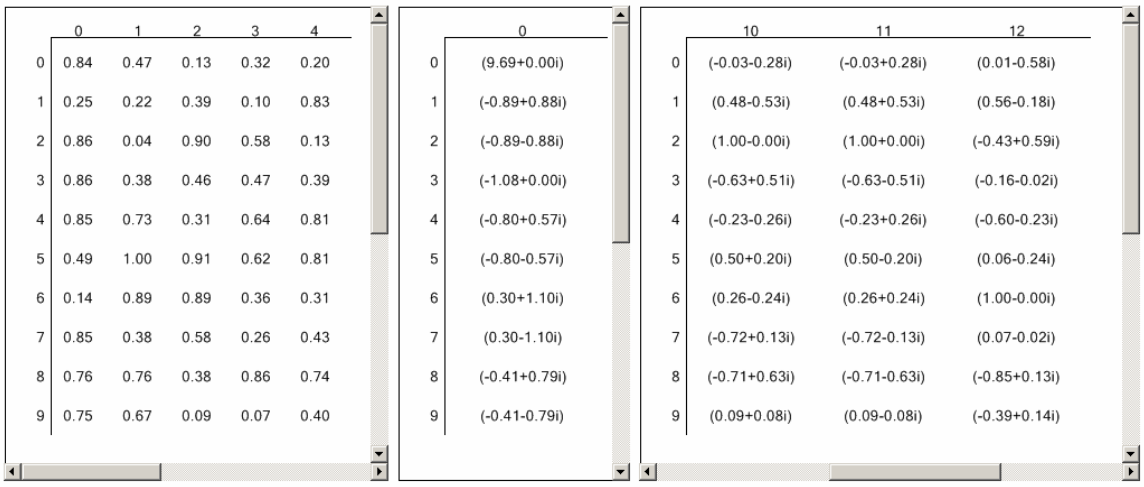
```
int n = 4096;
double [] xR = new double[n];
double [] iR = new double[n];
for (int i=0; i < n; i++)
{
  .
  .
  .
}

DDMat origData = new DDMat(xR);

// Since iR[i] is all zeros, can use real only version of DSP
DSP fftobj = new DSP(origData);
DDMat freqdata = new DDMat();
//This routine takes the raw data, pre-RealFFT
// sample interval for 1000.0 Hz is 0.001 seconds
DDMat powerSpectrum = fftobj.powerSpectrum( 0.001, freqdata);

)
```

10. Matrix Viewers

com.quinncurtis.matviewjava.MatViewer
MatViewerBase
DMatViewer
XMatViewer

The QCMatViewJava DLL contains a couple of simple Java controls, **DMatViewer** and **XMatViewer**, that display real and complex matrix data on a form in a scrollable matrix format. These controls are for use in Java **JPanel** based applets and applications. The matrix viewer size, number of rows, number of columns, fonts and numeric format are all user programmable. The matrix viewers can display a number of rows and columns only limited the amount of free memory accessible by the program. A matrix viewer is read-only by default, but it can be made editable using the **enableEdit** method.

DMatViewer XMatViewer XMatViewer



	0	1	2	3	4
0	0.84	0.47	0.13	0.32	0.20
1	0.25	0.22	0.39	0.10	0.83
2	0.86	0.04	0.90	0.58	0.13
3	0.86	0.38	0.46	0.47	0.39
4	0.85	0.73	0.31	0.64	0.81
5	0.49	1.00	0.91	0.62	0.81
6	0.14	0.89	0.89	0.36	0.31
7	0.85	0.38	0.58	0.26	0.43
8	0.76	0.76	0.38	0.86	0.74
9	0.75	0.67	0.09	0.07	0.40

	0
0	(9.69+0.00i)
1	(-0.89+0.88i)
2	(-0.89-0.88i)
3	(-1.08+0.00i)
4	(-0.80+0.57i)
5	(-0.80-0.57i)
6	(0.30+1.10i)
7	(0.30-1.10i)
8	(-0.41+0.79i)
9	(-0.41-0.79i)

	10	11	12
0	(-0.03-0.28i)	(-0.03+0.28i)	(0.01-0.58i)
1	(0.48-0.53i)	(0.48+0.53i)	(0.56-0.18i)
2	(1.00-0.00i)	(1.00+0.00i)	(-0.43+0.59i)
3	(-0.63+0.51i)	(-0.63-0.51i)	(-0.16-0.02i)
4	(-0.23-0.26i)	(-0.23+0.26i)	(-0.60-0.23i)
5	(0.50+0.20i)	(0.50-0.20i)	(0.06-0.24i)
6	(0.26-0.24i)	(0.26+0.24i)	(1.00-0.00i)
7	(-0.72+0.13i)	(-0.72-0.13i)	(0.07-0.02i)
8	(-0.71+0.63i)	(-0.71-0.63i)	(-0.85+0.13i)
9	(0.09+0.08i)	(0.09-0.08i)	(-0.39+0.14i)

Real Matrix Complex Matrix Complex Matrix

Typical use of the matrix viewers do not have you using the constructors. You would normally add a matrix viewer to a form by dropping it from the Toolbox, which automatically calls the default constructor.

DMatViewer constructors

```
public DMatViewer(
   DMatBase mat,
   int rows,
   int cols
);
```

XMatViewer constructors

```
public XMatViewer(
   XMatBase mat,
   int rows,
   int cols
);
```

mat	The source matrix. **DMatViewer** uses a **DDMat** matrix while **XMatViewer** uses a **DXMat matrix**
row	The number of display rows in the matrix viewer.
cols	The number of display columns in the matrix viewer.

If you add a matrix viewer to a form from the Toolbox, initialize the viewer matrix, rows and columns using the **SetViewMatrix** method.

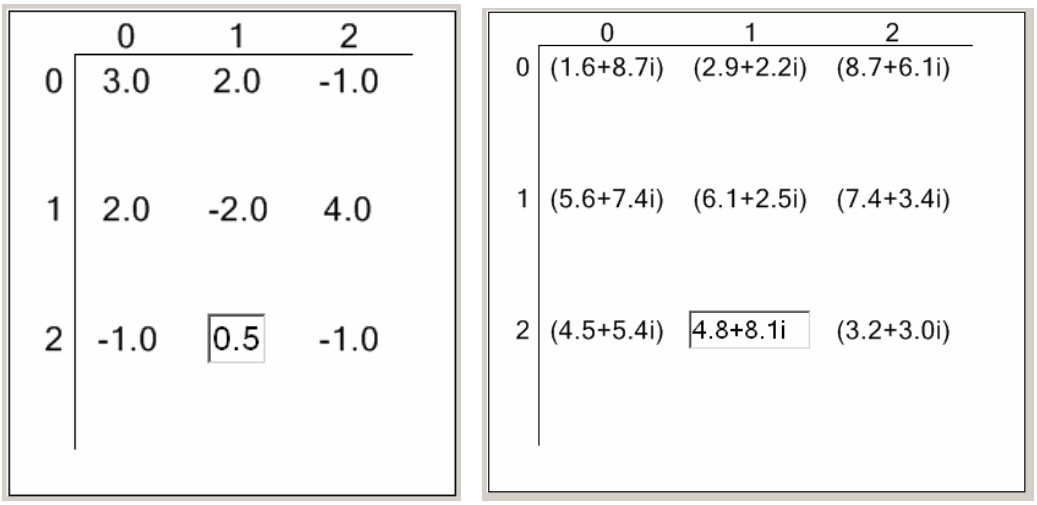
XMatViewer.setViewMatrix Method

```
public void setViewMatrix(
   DMatBase mat,
   int rows,
   int cols
);
```

mat	The source matrix. **DMatViewer** uses a **DDMat** matrix while **XMatViewer** uses a **DXMat matrix**
row	The number of display rows in the matrix viewer.
cols	The number of display columns in the matrix viewer.

Set the decimal precision of the numeric values in the matrix viewer using the **Decimals** property. If you want to be able to edit the individual cells of a matrix, call the **enableEdit(true)** method. If you make any changes to the properties of a matrix viewer, call the **updateDraw** method to force an immediate update. If the matrix viewer is edit enabled, click on the matrix element you want to edit. An edit box will appear in place of the element. Make your edit and hit return, or click anywhere else in the matrix viewer.

Any edits made to a matrix using a matrix viewer will immediately update the associated matrix; either a **DDMat** or **DXMat** object.



Above you see what the edit mode looks when the cell [2][1] is edited in real (DDMat) and complex (DXMat) matrices.

Example (Extracted from the EigenRealUnSym example program)

```
dMatViewer1 = new DMatViewer();
dMatViewer1.setPosition(40,100,300,296);
dMatViewer1.setViewMatrix(A, 8, 5);
dMatViewer1.setDecimals( 2);
this.add(dMatViewer1);.
.
.
xMatViewer1.setViewMatrix(EValues,10,1);
xMatViewer2.setViewMatrix(EVectors,10,3);

xMatViewer1.setDecimals( 2);
xMatViewer2.setDecimals(2);

xMatViewer1.updateDraw();
xMatViewer2.updateDraw();
```

11. Thermocouple Linearization

TCLin

Thermocouples are the most widely used transducers for measuring temperature. They are economical, rugged and have stable characteristics over years. Further, available types can measure temperatures ranging from the cryogenic to jet exhaust.

Thermocouple devices are two pieces of metal in isothermal contact (physical contact at the same temperature). Because the number of free electrons in a metal depends on its temperature and composition, thermocouples exhibit a voltage potential that's a repeatable function of temperature. Over the past hundred years, several metallic pairs have become standard thermocouple types because of their accuracy and usefulness across common temperature ranges.

THERMOCOUPLES

Type	Composition	Temperature range, °
B	Pt-30% Rh versus Pt-6% Rh	0 to 1820
E	Ni-Cr alloy versus a Cu-Ni alloy	-270 to 1000
J	Fe versus a Cu-Ni alloy	-210 to 1200
K	Ni-Cr alloy versus Ni-Al alloy	-270 to 1372
N	Ni-Cr-Si alloy versus Ni-Si-Mg alloy	-270 to 1300
R	Pt-13% Rh versus Pt	-50 to 1768
S	Pt-10% Rh versus Pt	-50 to 1768
T	Cu versus a Cu-Ni alloy	-270 to 400

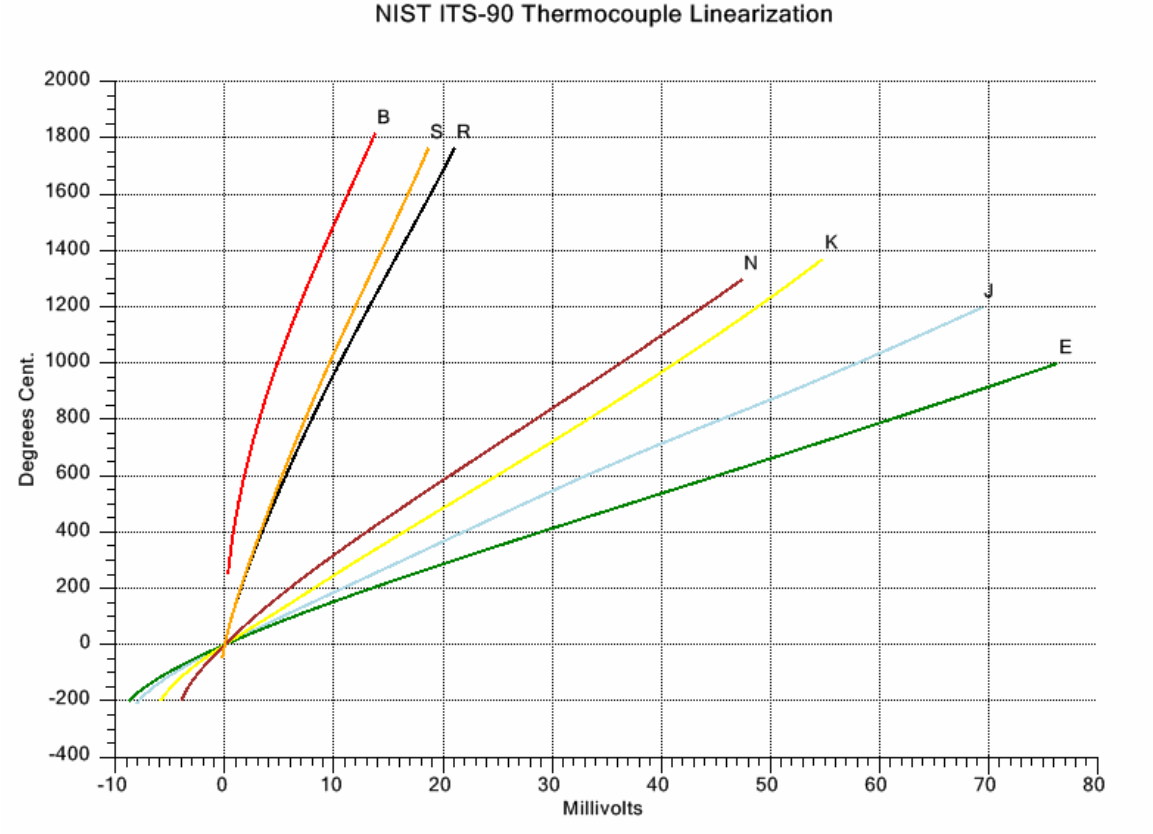
NIST ITS-90

The relationship between a thermocouple's output voltage and its temperature is a nonlinear function best approximated with a high-order polynomial. Once you measure a thermocouple's output voltage you can calculate a temperature by plugging the voltage into a polynomial equation or by looking up the voltage in standard thermocouple tables published by the National Bureau of Standards (NBS).

The thermocouple routines in this package are based on the reference tables summarized in NIST ITS-90 as published in the NIST Monograph 175. This standard defines 8 standard thermocouple types: B, E, J, K, N, R, S, T. Each of the 8 thermocouple types have one or more polynomial equations which define thermocouple voltage as a function of temperature.

It is important to understand that the thermocouple temperature to voltage tables presented in NIST ITS-90 are not from direct measurements. Thermocouple voltages were sampled at discrete intervals of temperature and equations fitted to these sampled

points. The NIST ITS-90 tables were generated from the fitted equation and not actual measured values.



The polynomial equations used to convert from volts to temperature, and temperature to volts, are those specified in the NIST ITS-90 tables. The implicit errors for the voltage to temperature conversions are listed below:

Temperature Sub range

	#1	#2	#3	#4
Thermocouple Type:	**J**			
Temperature Range:	-210 °C to 0 °C	0 °C to 760 °C	760 °C to 1200 °C	
Voltage Range:	-8.095 mV to 0 mV	0 mV to 42.919 mV	42.919 mV to 69.553 mV	
Error Range:	-0.05 °C to 0.03 °C	-0.04 °C to 0.04 °C	-0.04 °C to 0.03 °C	
Thermocouple Type:	**B**			
	250 °C to 700 °C	700 °C to 1820 °C		
Voltage Range:	0.291 to 2.431mV	2.431 mV to 13.820 mV		
Error Range:	-0.02 °C to 0.03 °C	-0.01 °C to 0.02 °C		

Thermocouple Type:	**E**	
	-200 °C to 0 °C	0 °C to 1000 °C
Voltage Range:	-8.825 mV to 0 mV	0 mV to 76.373 mV
Error Range:	-0.01 °C to 0.03 °C	-0.02 °C to 0.02 °C

Thermocouple Type:	**K**		
Temperature Range:	-200 °C to 0 °C	0 °C to 500 °C	500 °C to 1372 °C
Voltage Range:	-5.891 mV to 0 mV	0 mV to 20.644 mV	20.644 mV to 54.886 mV
Error Range:	-0.02 °C to 0.04 °C	-0.05 °C to 0.04 °C	-0.05 °C to 0.06 °C

Thermocouple Type:	**N**		
Temperature Range:	-200 °C to 0 °C	0 °C to 600 °C	600 °C to 1300 °C
Voltage Range:	-3.990 to 0 mV	0 to 20.613 mV	20.613 to 47.513 mV
Error Range:	-0.02 °C to 0.03 °C	-0.02 °C to 0.03 °C	-0.04 °C to 0.02 °C

Thermocouple Type:	**R**			
Temperature Range:	-50 °C to 250 °C	250 °C to 1200 °C	1064 °C to 1664.5 °C	1664.5 °C to 1768.1 °C
Voltage Range:	-0.226 to 1.923 mV	1.923 to 13.228 mV	11.361 to 19.739 mV	19.739 to 21.103 mV
Error Range:	-0.02 °C to 0.02 °C	-0.005 °C to 0.005 °C	-0.0005 °C to 0.001 °C	-0.001 °C 0.002 °C to

Thermocouple Type:	**S**			
Temperature Range:	-50 °C to 250 °C	250 °C to 1200 °C	1064 °C to 1664.5 °C	1664.5 °C to 1768.1 °C
Voltage Range:	-0.235 to 1.874 mV	1.874 to 11.950 mV	10.332 to 17.536 mV	17.536 to 18.693 mV
Error Range:	-0.02 °C to 0.02 °C	-0.01 °C to 0.01 °C	-0.0002 °C to 0.0002 °C	-0.002 °C to 0.002 °C

Thermocouple Type:	**T**	
Temperature Range:	-200 °C to 0 °C	0 °C to 400 °C
Voltage Range:	-5.603 mV to 0 mV	0 mV to 20.872 mV
Error Range:	-0.02 °C to 0.04 °C	-0.03 °C to 0.03 °C

There are no errors in the temperature to voltage conversion conversions. Our routines use the temperature to voltage polynomials specified in NIST ITS-90. Those polynomials *define* the voltage to temperature curves. All errors are measured with respect to those curves.

Cold Junction Compensation

When making thermocouple measurements you must take into account the effect of the temperature of the terminal block, terminating the non-hot end of the thermocouple. This is referred to as cold-junction-compensation. Typically, this involves making a temperature measurement of the terminal block, using a non-thermocouple based measuring device, usually a solid-state temperature sensor (the Analog Devices AD590) or a precision thermistor. The terminal block temperature is converted to an equivalent

thermocouple voltage (use our **tempToMilliVolts** method) and the resulting voltage added to the measured thermocouple voltage. The *compensated* voltage, now a sum of the thermocouple voltage and the equivalent junction block voltage, is what you actually convert back to temperature using our **milliVoltsToTemp** method. If you are not familiar with cold-junction-compensation, here is a good reference for you to study: http://www.maxim-ic.com/appnotes.cfm/an_pk/4026.

Class TCLin

The TCLin class provides methods to convert thermocouple millivolts into equivalent temperatures, and thermocouple temperatures into equivalent millivolts. The later conversion (temperature to millivolts) can be used in place of the Seebeck coefficient for use in the thermocouples cold-junction compensation calculations.

TCLin constructor

This constructor initializes the class for a specific thermocouple type. Use one of the thermocouple, TCLin .TC_TYPE type constants: TC_TYPE_J, TC_TYPE_B, TC_TYPE_E, TC_TYPE_K, TC_TYPE_N, TC_TYPE_R, TC_TYPE_S, TC_TYPE_T.

```
public TCLin(
   int tct
);
```

Parameters

tct Thermocouple type. Use one of the thermocouple, TCLin .TC_TYPE type constants: TC_TYPE_J, TC_TYPE_B, TC_TYPE_E, TC_TYPE_K, TC_TYPE_N, TC_TYPE_R, TC_TYPE_S, TC_TYPE_T.

Once the class is instantiated for a specific thermocouple, call the **milliVoltsToTemp** method to convert thermocouple millivolts into a temperature (degrees Centigrade) and **tempToMilliVolts** to convert a thermocouple temperature into millivolts (used in cold-junction compensation).

Method TCLin.milliVoltsToTemp

Convert thermocouple millivolts to temperature in degrees Centigrade.

```
public double milliVoltsToTemp(
   double mv
);
```

Parameters

mv Thermocouple voltage in millivolts.

Return Value

Returns the thermocouple temperature in degrees Centigrade.

Method TCLin.milliVoltsToTempIterative

Convert thermocouple millivolts to temperature in degrees Centigrade. This is the ultimate in voltage to temperature conversion for a thermocouple. It first uses the **MilliVoltsToTemp** method to provide an initial estimate. Then it backward iterates on the on the temperature to millivolts polynomials, for which there is zero error, using the Newton-Raphson of root solving. This ends up by returning a millivolts to temperature conversion accurate to at least the specified tolerance.

```
public double milliVoltsToTempIterative(
   double mv
   double tolerance
);
```

Parameters

mv Thermocouple voltage in millivolts.

tolerance Iterations are carried out until the tolerance level is reached. Because the Newton-Raphson method converges extremely fast, typically the accuracy of the conversion is 3 – 6 orders of magnitude more accurate than the tolerance. The minimum tolerance that you can specify is 10e-10.

Return Value

Returns the thermocouple temperature in degrees Centigrade.

Method TCLin.tempToMilliVolts

Convert thermocouple temperature to millivolts.

```
public double TempToMilliVolts(
   double temp
);
```

Parameters

temp Thermocouple temperature in degrees Centigrade.

Return Value

Returns the thermocouple voltage in millivolts.

There are also simple methods (**getMinMilliVolts, getMaxMilliVolts, getMinTemp,** and **getMaxTemp**) that return the minimum and maximum voltages and temperatures associated with a given thermocouple type.

Example

The example program TCLinTest divides each thermocouple voltage range into 10,000 equal segments and calculates the **milliVoltsToTemp** conversion for each point. It then feeds the resulting temperature back to the **tempToMilliVolts** method, and compares the resulting millivolt measurement to the original and remembers the maximum error. This represents a good test of the overall accuracy of the linearization routines, since the **milliVoltsToTemp** and **tempToMilliVolts** methods used different linearization polynomials.

```
int tci = TCLin.TC_TYPE_J;

TCLin tc = new TCLin( tci);
double mV= tc.getMinMilliVolts(), mv2 = 0.0;
double temp = 0.0;
int count = 10000;
double voltageInc = (tc.getMaxMilliVolts() - tc.getMinMilliVolts())/count;
double maxerror = 0.0, error = 0;
for (int i=0; i < count; i++)
{
    temp = tc.milliVoltsToTemp(mV);
    mv2 = tc.tempToMilliVolts(temp);
    error = Math.abs(mV - mv2);
    if (error > maxerror)
        maxerror = error;
    if (error >0.1)
        break;
    mV += voltageInc;
}
```

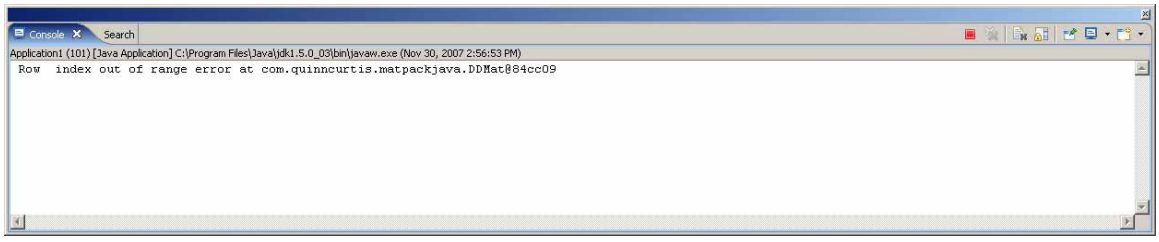
A more practical method would simulate an actual thermocouple measurement, with cold-junction-compensation.

```
// This would be replaced by your own routine that reads uses a voltmeter or A/D
// converter to read the thermocouple voltage.
double getThermocoupleMilliVolts()
{
    double tcmv = 10.1; //millivolts
    return tcmv;
}

// This would be replaced by your own routine that reads uses a voltmeter or A/D
// converter to read the cjc temperature.
double getCJCTemperature()
{
    double cjctemp = 16.3; //degrees Cent.
    return cjctemp;
}

public double tCLinearizeWithCJC(int tci)
{
    int tct = TCLin.TC_TYPE_K;

    TCLin tc = new TCLin( tct);
    // You would need to implement the routine GetThermocoupleMilliVolts();
    double tcMilliVolts = getThermocoupleMilliVolts();
    // You would need to implement the routine GetCJCTemperature ();
    double cjcTemp = getCJCTemperature();

// Convert terminal block temperature to equivalent thermocouple millivolts
    double cjcEquivMilliVolts = tc.tempToMilliVolts(cjcTemp);

    // Add the cjc millivolts to the thermocouple millivolts
    tcMilliVolts += cjcEquivMilliVolts;
    // Convert the cjc compensated millivolts to temperature
    double tcTemperature = tc.milliVoltsToTemp(tcMilliVolts);
    return tcTemperature;
}
```

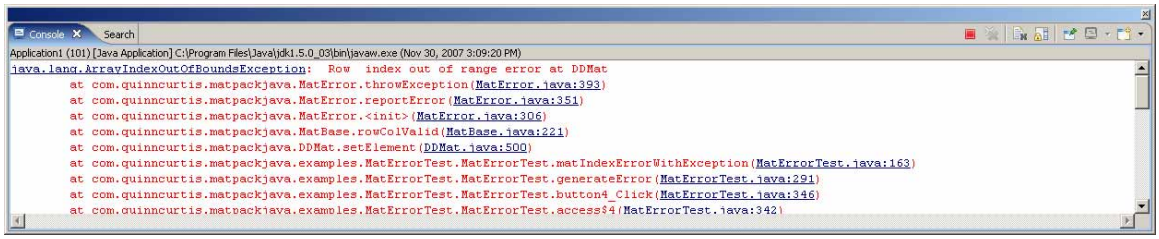
12. Error Handling

The matrix math classes and numerical algorithm classes can generate a limited number of runtime errors. Any time an error is detected, an object **MatError** object is created, and that object processes the error. The **MatError** object can process the error in several different ways. It can print an error message to a message box on the screen,

print an error message to the system console,


Console
Search
Application1 (101) [Java Application] C:\Program Files\Java\jdk1.5.0_03\bin\javaw.exe (Nov 30, 2007 2:56:53 PM)
Row index out of range error at com.quinncurtis.matpackjava.DDMat@84cc09

throw and exception which you can trap and process in your own way,



ignore the error, or exit the program.

The standard errors that the software generates are listed below:

```
case NOMEM: throw new java.lang.OutOfMemoryError(sMessageString);

case NULLMAT: throw new java.lang.NullPointerException(sMessageString);


case INVALID_ARG_DIM: throw new java.lang.IllegalArgumentException(sMessageString);

case INVALID_MAT_DIM: throw new java.lang.IllegalArgumentException(sMessageString);


case SYMMETRIC_MATRIX_REQUIRED: throw new java.lang.IllegalArgumentException(sMessageString);

case POSDEFINITE_MATRIX_REQUIRED: throw new
java.lang.IllegalArgumentException(sMessageString);

case HERMITIAN_MATRIX_REQUIRED: throw new java.lang.IllegalArgumentException(sMessageString);
```

Error Constant	Description	Exception (if enabled)
NOERR	No error	
MAT_COL_ERR	Column index out of range error	java.lang.ArrayIndexOutOfBoundsException
MAT_ROW_ERR	Row index out of range error	java.lang.ArrayIndexOutOfBoundsException
MAT_ELEMENT_ERR	Array Element Error;	java.lang.ArrayIndexOutOfBoundsException
INVALID_INDEX	Array Index Exceeds Dimension	java.lang.ArrayIndexOutOfBoundsException
NOMEM	Not Enough Memory	java.lang.OutOfMemoryError
NULLMAT	Null matrix argument error	java.lang.NullPointerException
INVALID_ARG_DIM	Dimensions Do Not Match	java.lang.IllegalArgumentException
INVALID_MAT_DIM	Matrix should have the same number of rows and columns	java.lang.IllegalArgumentException
SYMMETRIC_MATRIX_REQUIRED	Matrix must be symmetric, ie. A = Transpose(A)	java.lang.IllegalArgumentException
POSDEFINITE_MATRIX_REQUIRED	Matrix must be positive definite.	java.lang.IllegalArgumentException
HERMITIAN_MATRIX_REQUIRED	Matrix must be Hermitian	java.lang.IllegalArgumentException
NO_DSPACE	Insufficient Disk Space	java.lang.RuntimeException
FILE_ERROR	File Read/Write Error	java.lang.RuntimeException
INVALID_FILENAME	Invalid file name	java.lang.RuntimeException
SINGULAR	Matrix Is Singular	Does not throw and exception
NO_ACC	Solution Is Not Accurate	Does not throw and exception

Select the error processing method that you want by setting the static MatError.setReportMode method to one of the report mode constants:

```
public final static int       ErrorReportModes_DO_NOTHING = 0;

public final static int       ErrorReportModes_MESSAGE_BOX=1;
```

```
    public final static int       ErrorReportModes_SYSTEM_CONSOLE=2;

    public final static int       ErrorReportModes_THROW_EXCEPTION=3;

    public final static int       ErrorReportModes_EXIT=4;
```

The default is error reporting mode is MatError.ErrorReportModes_MESSAGE_BOX

The example below is extracted from the MatErrorTest example program and demonstrates the use of runtime exception error processing using try/catch blocks.

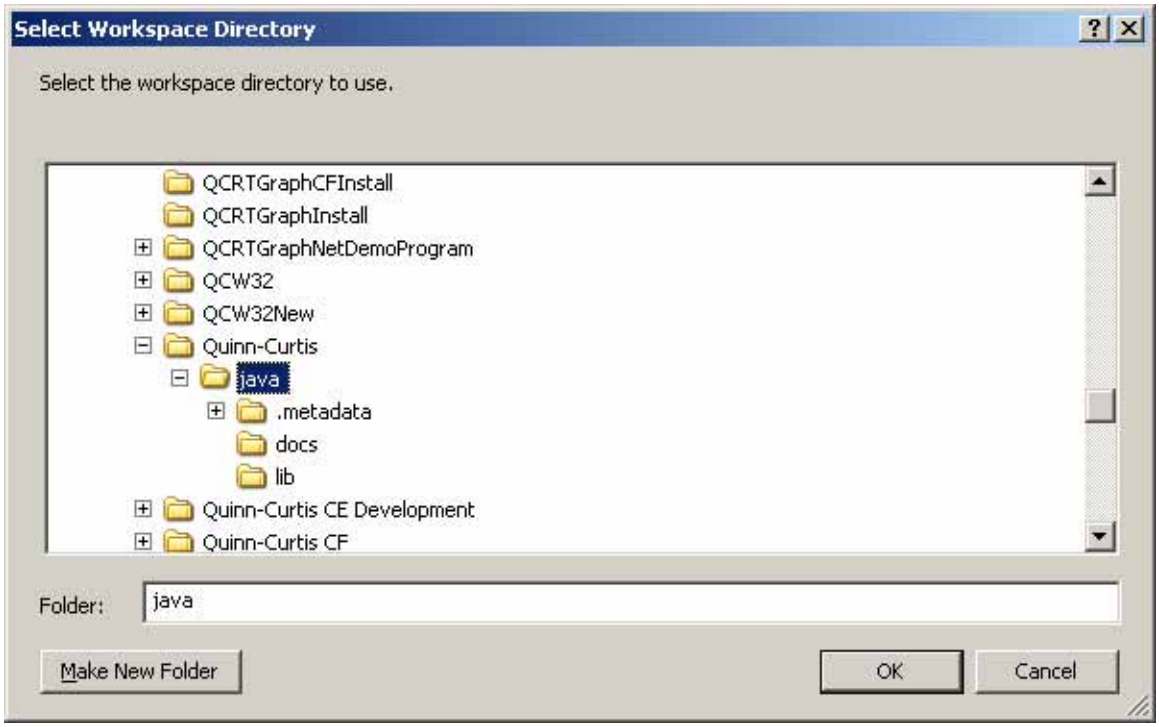
```
public void matIndexErrorWithException()
{
    try
    {
        int m = 2, n = 20;
        DDMat A = new DDMat(m, n);
        for (int i=0; i < m; i++)
            for (int j=0; j < n; j++)
                A.setElement(i,j, i + n + j);
        A.setElement(2,12,22); // generates a row index out of range error
    }
    catch (java.lang.ArrayIndexOutOfBoundsException exp)
    {
        exp.printStackTrace();
    }
}
```

13. Using QCMatPack for Java to Create Applications and Applets

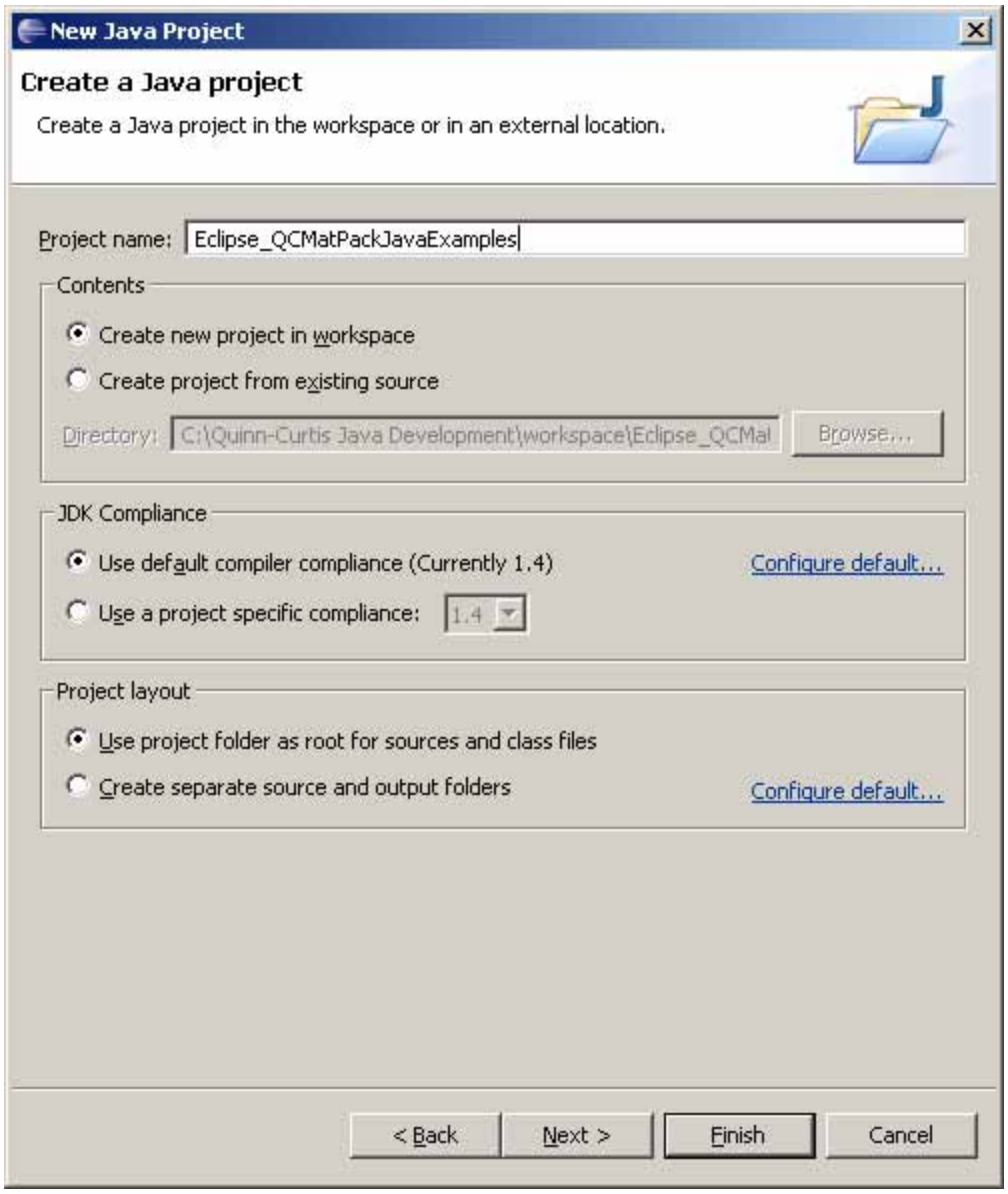
Eclipse

These directions were created using Eclipse Platform Version 3.0.2 . Our standard development environment is Eclipse and our example programs are organized along the lines of the Java example programs that ship with Eclipse.

From the Eclipse Java Perspective IDE, select **File | Switch Workspace** and browse to and select the **C:\Quinn-Curtis\java** directory.



This will establish the current workspace directory as Quinn-Curtis/java. From the Eclipse main menu select **File | New | Project | Java Project.** Give the project the name Eclipse_QCMatPackJavaExamples (any name will work).


New Java Project
Create a Java project
Create a Java project in the workspace or in an external location.
Project name: Eclipse_QCMatPackJavaExamples
Contents
Create new project in workspace
Create project from existing source
Directory: C:\Quinn-Curtis Java Development\workspace\Eclipse_QCMat
Browse...
JDK Compliance
Use default compiler compliance (Currently 1.4)
Configure default...
Use a project specific compliance: 1.4
Project layout
Use project folder as root for sources and class files
Create separate source and output folders
Configure default...
< Back
Next >
Finish
Cancel

Select **Next** to define the Java build settings. Select the **Libraries** tab and select **Add External JARs** and browse to the Quinn-Curtis/java/lib directory and add the qcmatpackjava.jar and qcmatviewjava.jar files.

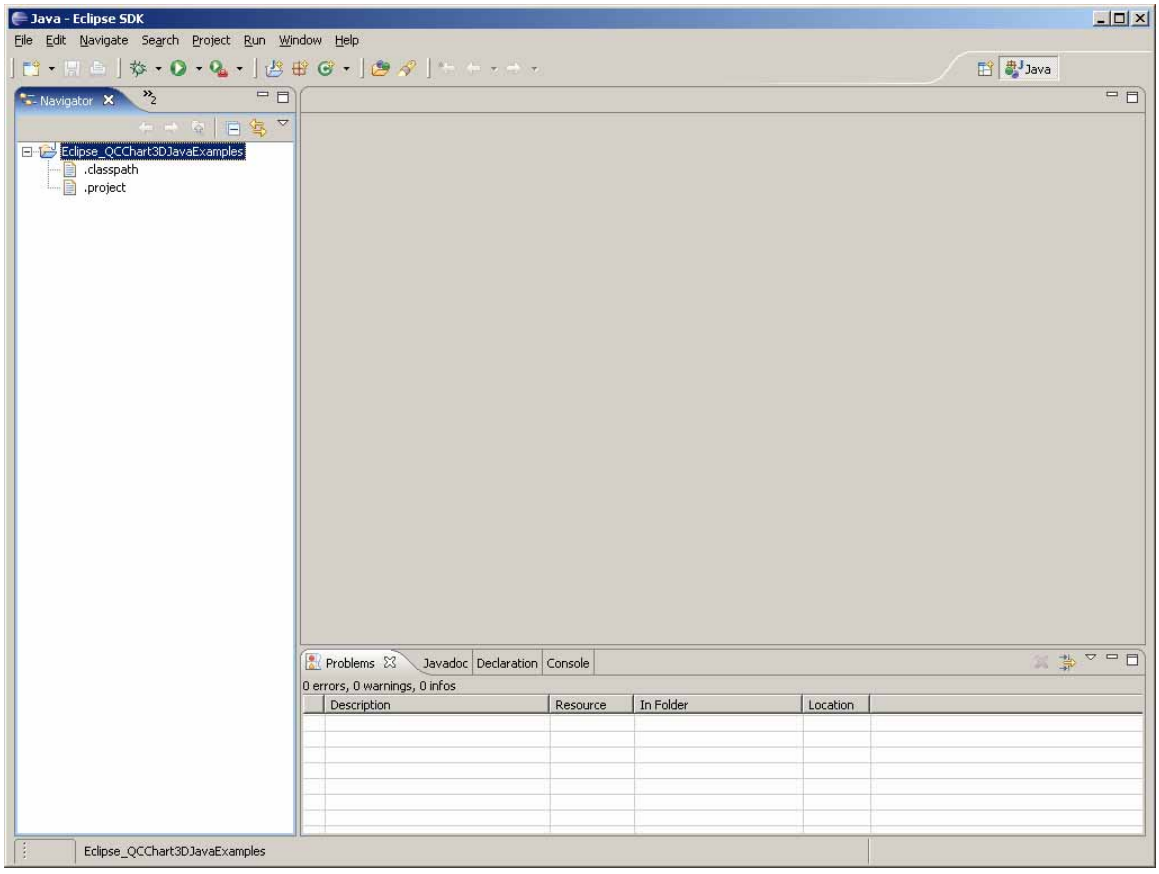
All of the example programs reference the QCMatPack and QCMatView for Java jar files (qcmatpackjava.jar and qcmatviewjava.jar) located in the Quinn-Curtis/java/lib folder. The qcmatpackjava.jar file contains the **com.quinncurtis.matpackjava** package reference in the

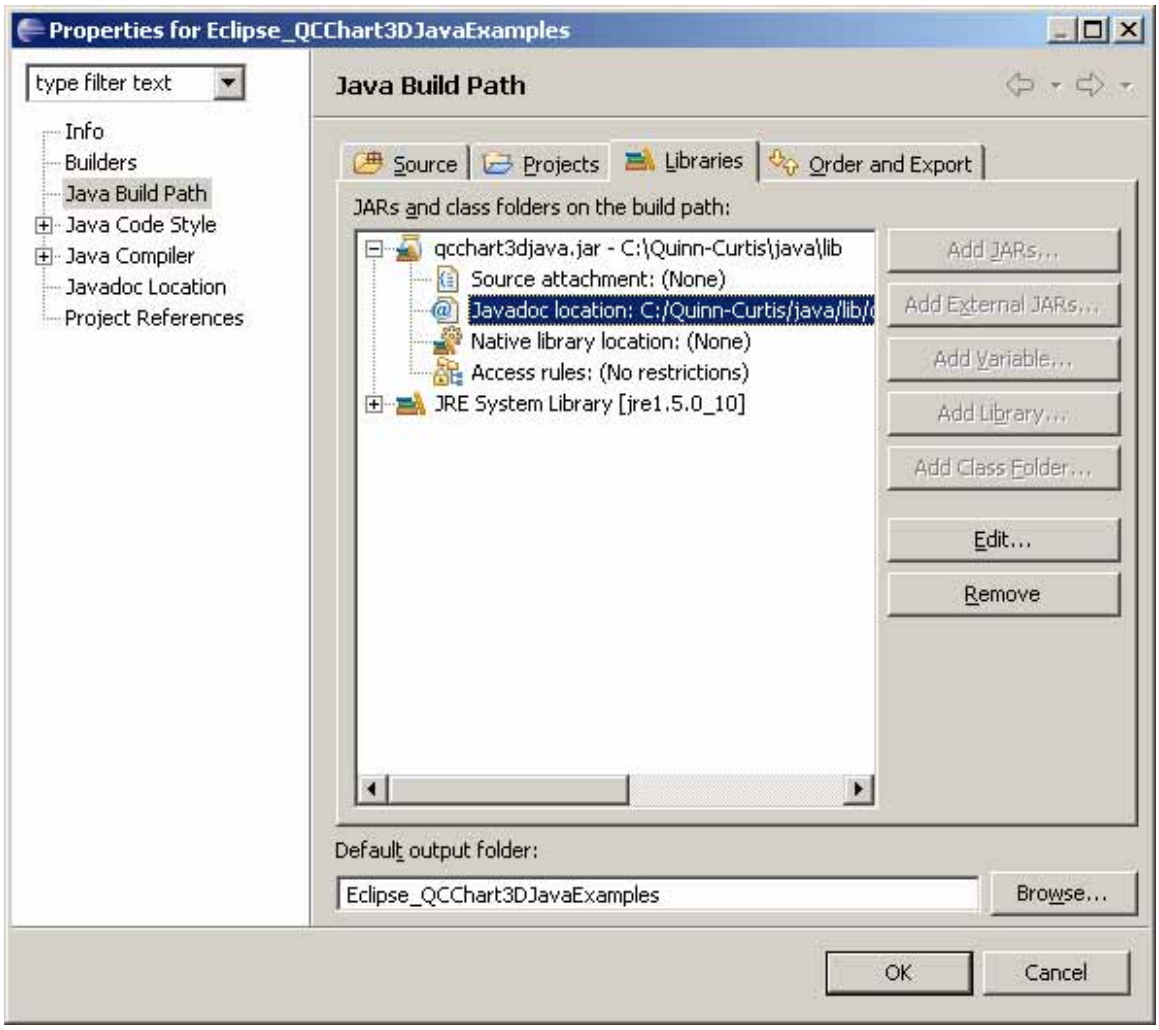
```
import com.quinncurtis.matpackjava.*;
```

statements found in all of the example program source files that contain calls to the charting routines.. The qcmatviewjava.jar file contains the com.quinncurtis.matviewjava package referenced in the

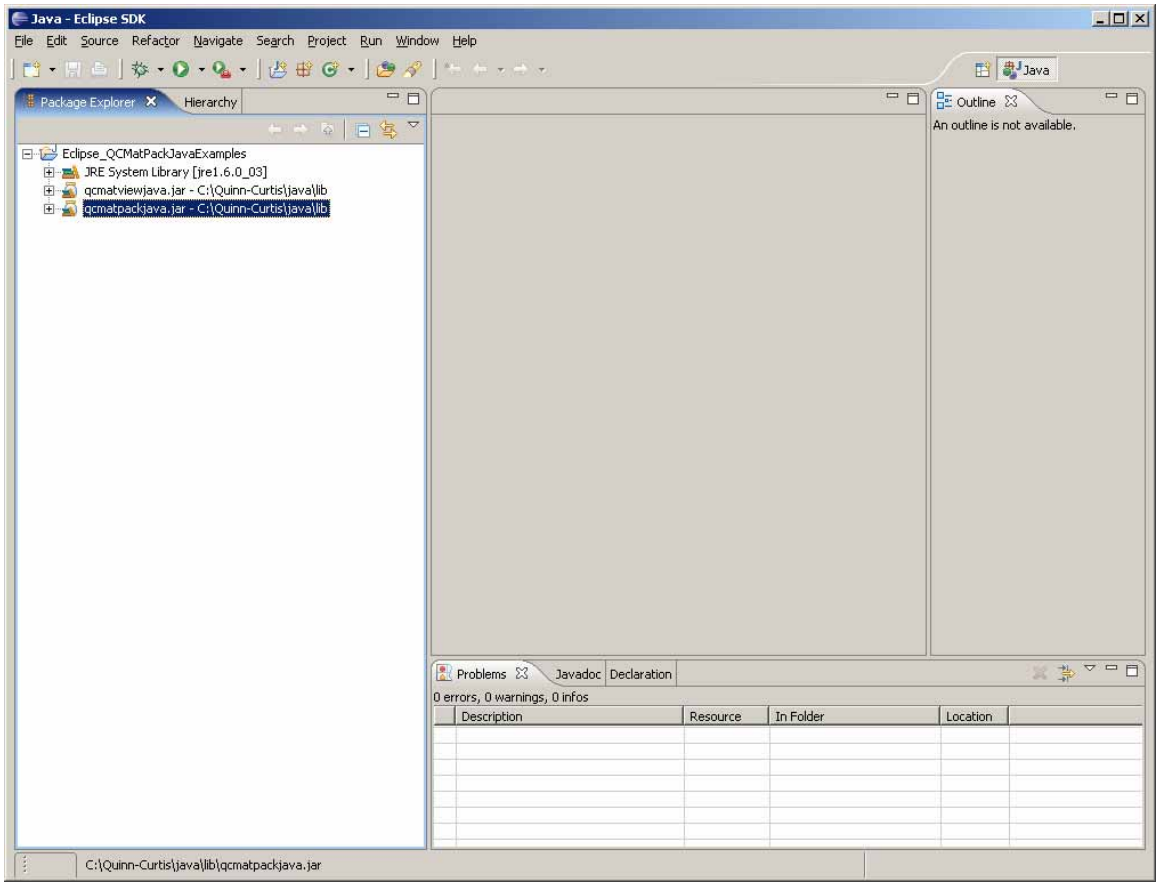
```
import com.quinncurtis.matviewjava.*;
```

Select **Finish** to complete the creation of the basic project settings. Say Yes if it asks you for the Perspective Switch. The Package Explorer should look like:

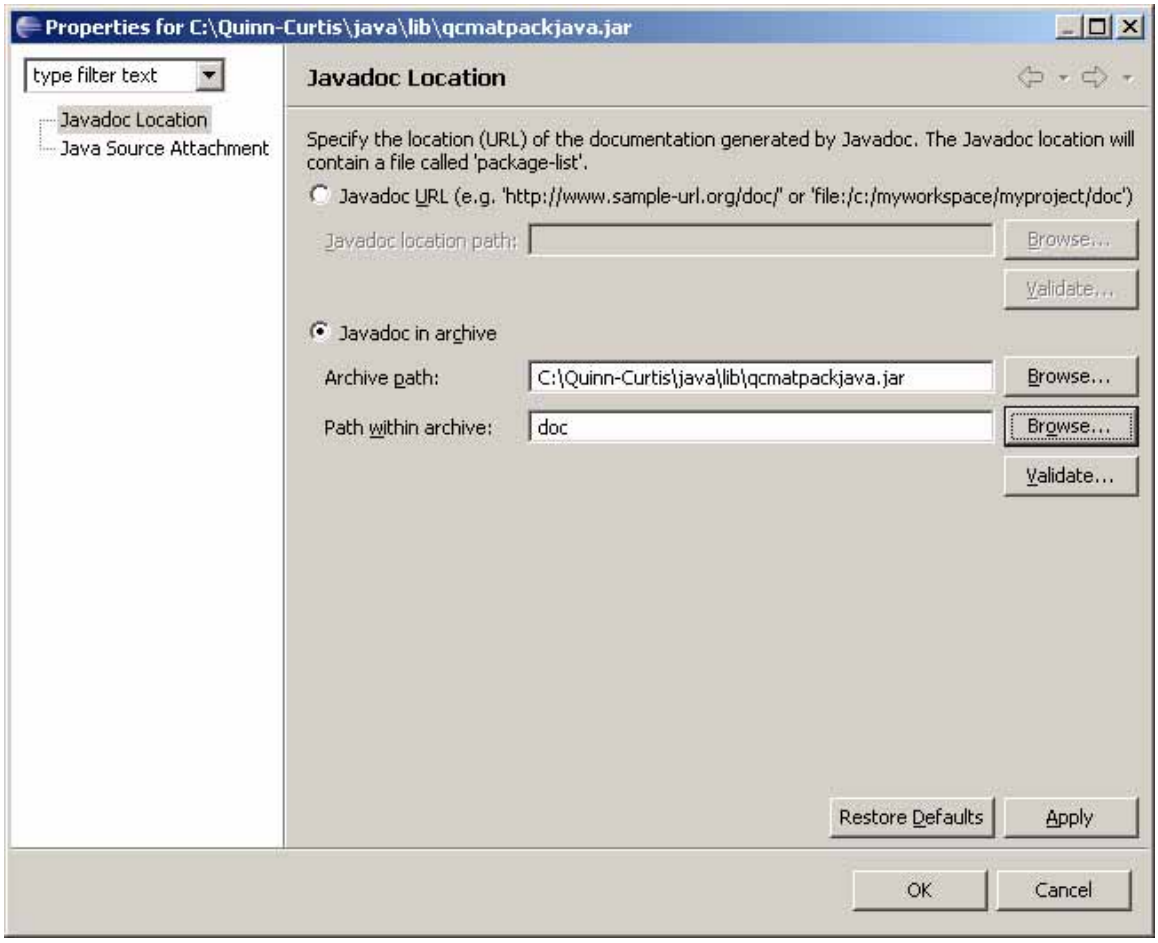




To add the QCMatPack Javadoc documentation to the project: expand the **Eclipse_QCMatPackJavaExamples** node and right click on **QCMatPackJava.jar** and select **Properties.**



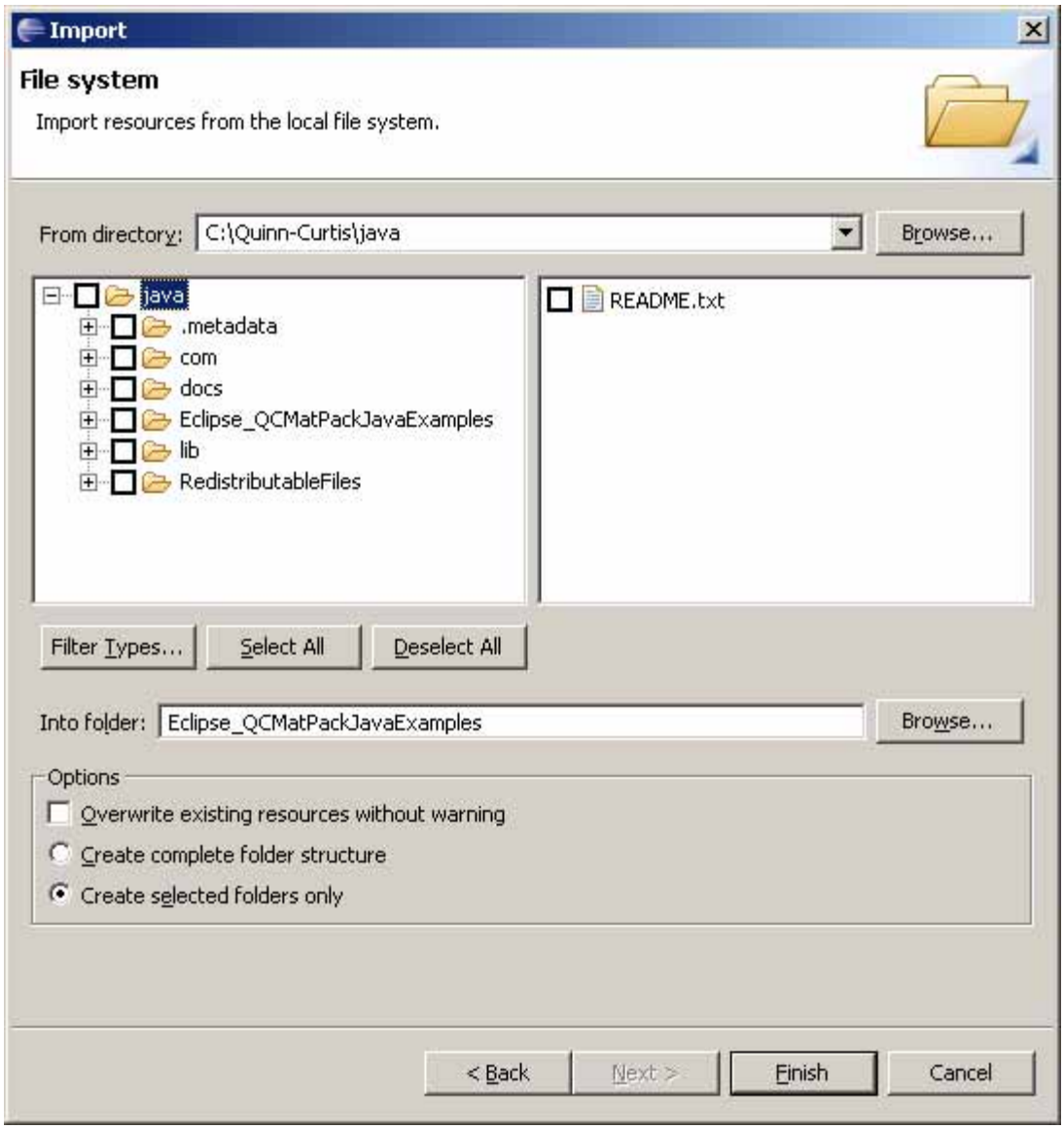
Select **Javadoc Location.** Select the **Javadoc in Archive** radio button, and browse to qcmatpackjava.jar file as the **Archive Path**, then select **doc** as the **Path within Archive**.



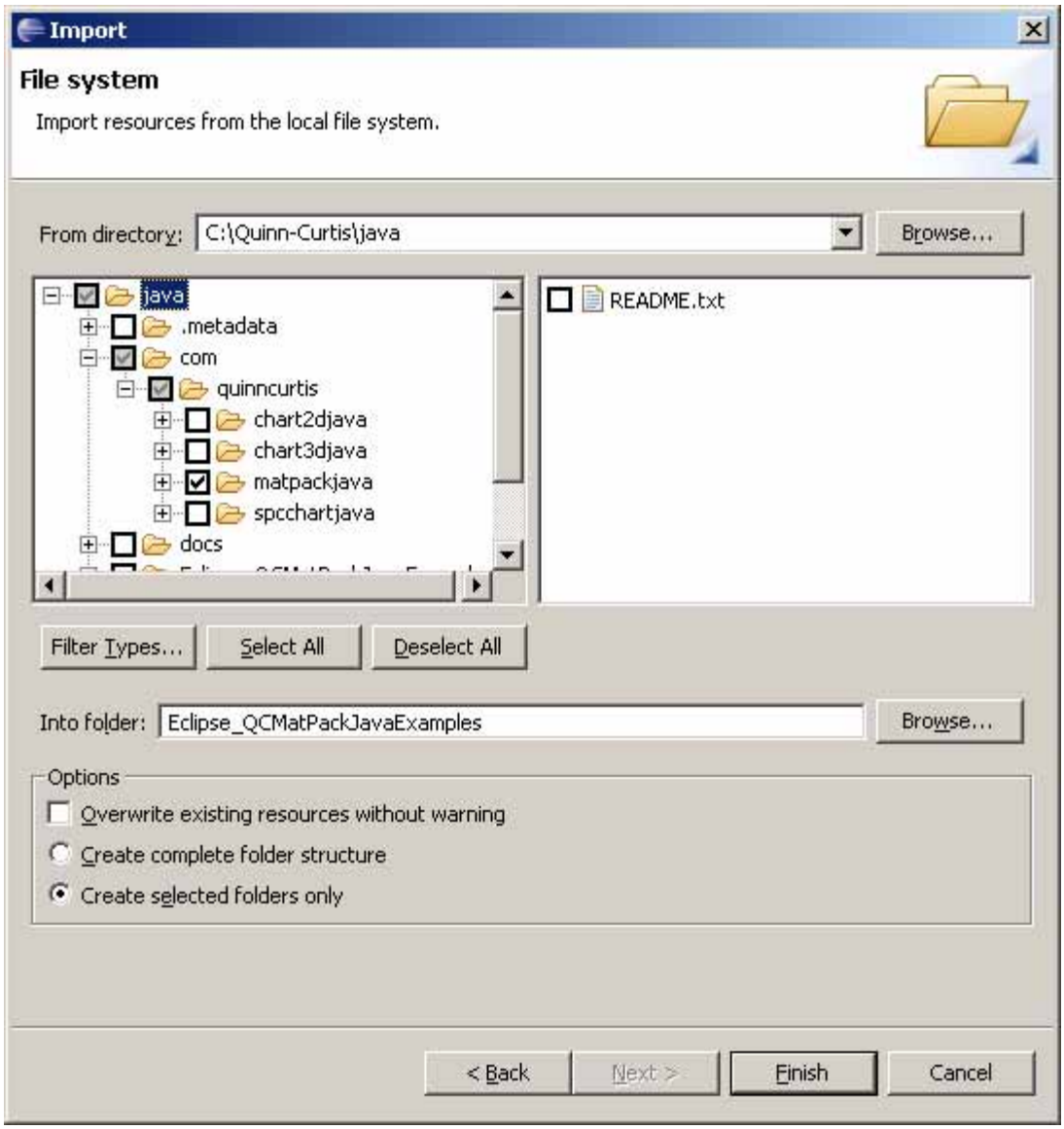
From that point on Eclipse F1 and Shift-F2 help should be available on the QCMatPack Java classes.

The project is still without the source files of the QCMatPack Java examples and these are added next.

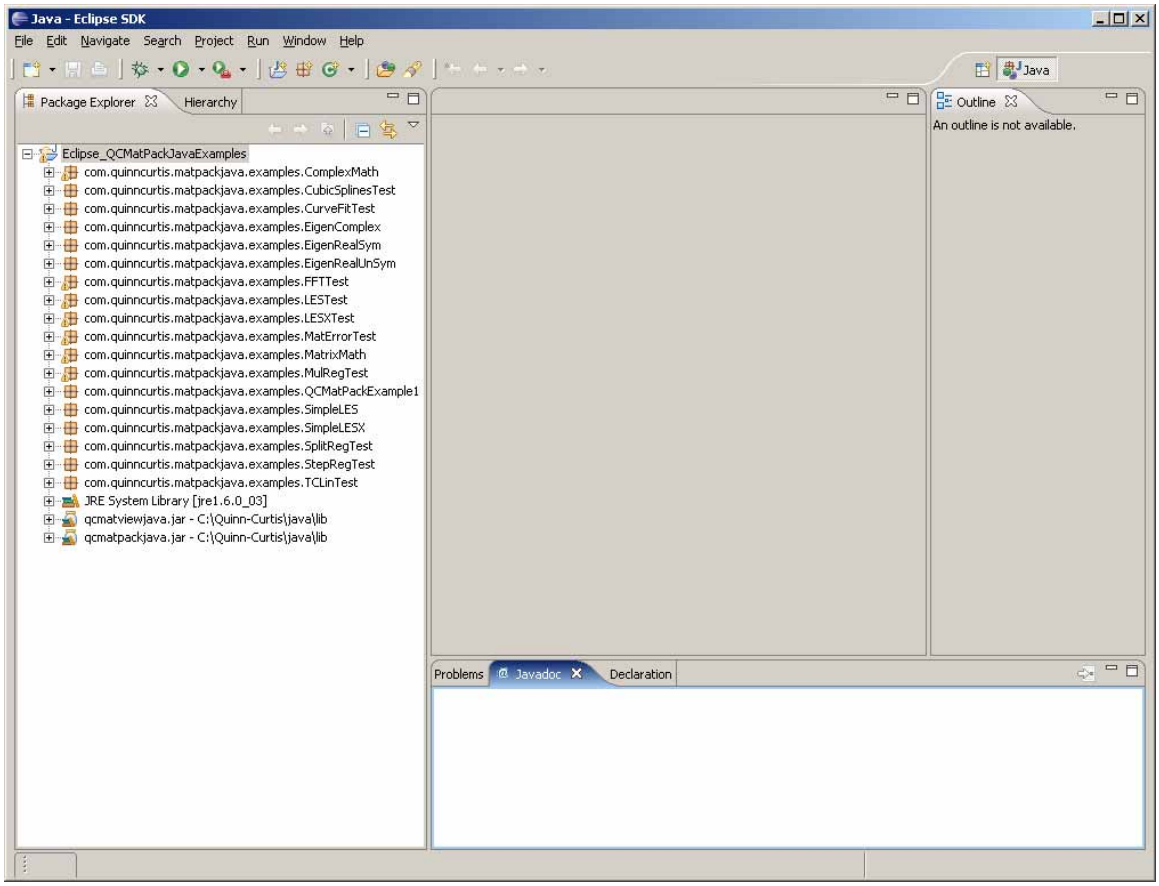
Right click the Eclipse_QCMatPackJavaExamples project in the Package Explorer and select **Import**. The Into Folder field should say QCMatPackJavaExamples. Select File System and Browse to the Quinn-Curtis/java directory.



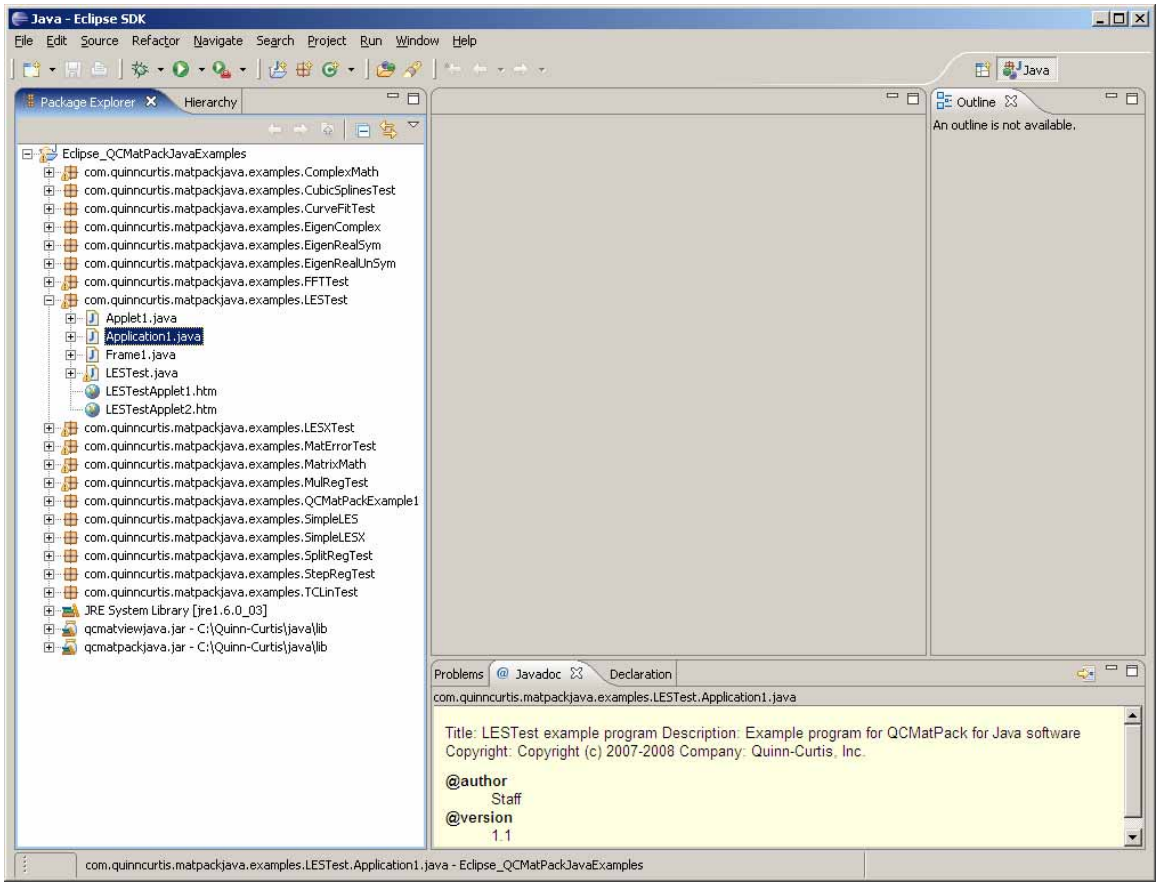
Expand the Quinn-Curtis/java directory and check the com/quinncurtis/matpackjava folder.



Select **Finish** and the example program folders under com/quinncurtis/matpackjava/examples will be added to the project. The entire contents of the example program folders will be copied so com/quinncurtis/matpackjava/examples directory structure is recreated under the Eclipse_QCMatPackJavaExample folder.



Expand one of the example program folders and locate the Application1.java file. Right click on that file and select **Run | Run Java Application**.



Java Applets

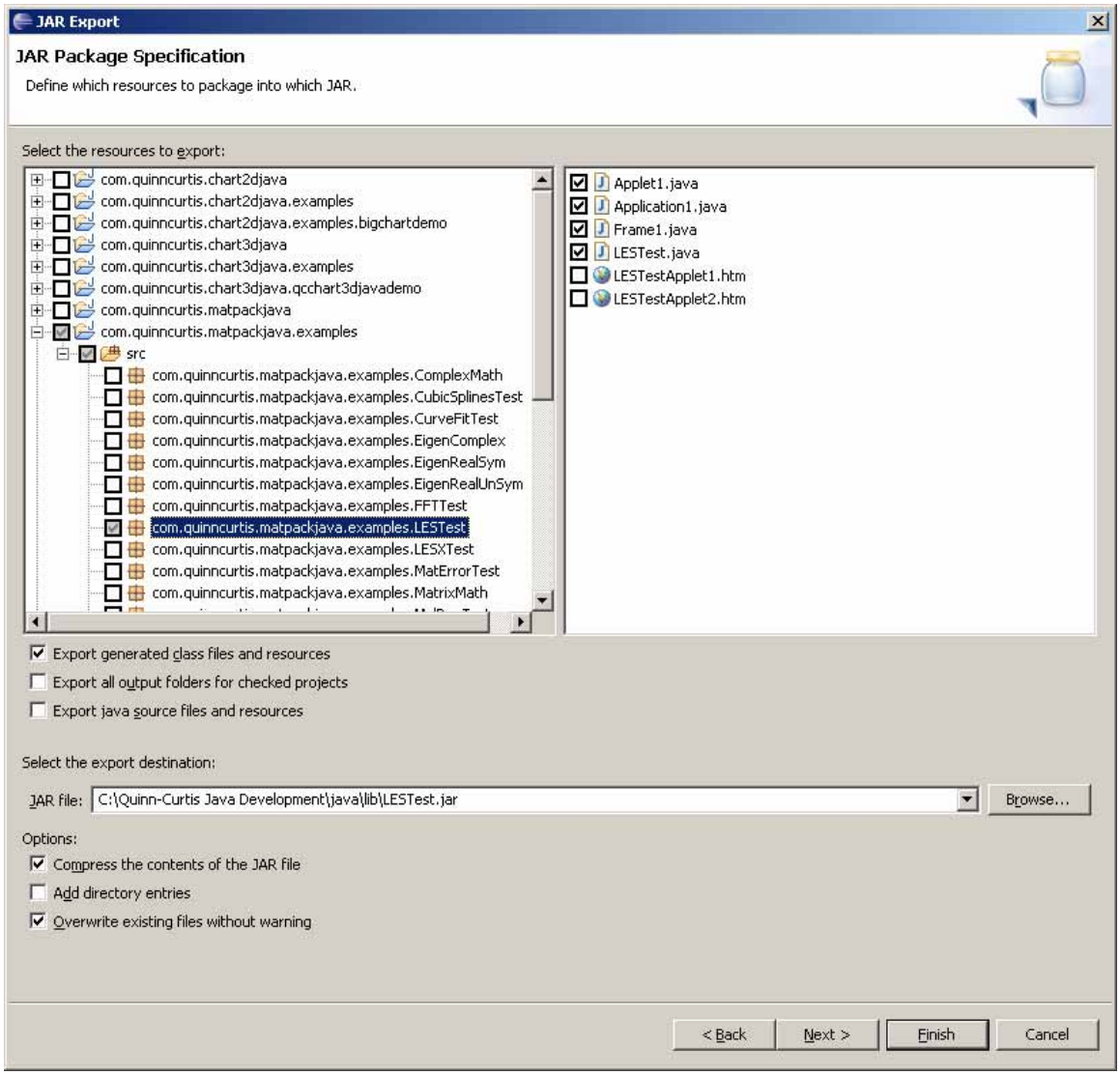
While Java Applications are useful if you are creating a program to run as a standalone application on a desktop, Java Applets are used to create Java programs that run in a web browser. All of the example programs include both application and applet versions. If you want to run one of the examples as an applet, run the Applet1.java file in the example programs folder. For Eclipse, right click on the Applet1.java file in the Package Explorer and choose Run As .. Java Applet.

In a Java applet, the **JFrame** window used in the Application file example is not, since that would force the graph to be created in a window separate from the browser. Instead, the example program JPanel is created in the content pane of the Applet window. The Applet1.java file below is extracted from the LESTest example program.

```
package com.quinncurtis.matpackjava.examples.LESTest;

import java.awt.*;

import javax.swing.JPanel;
```

```
public class Applet1 extends javax.swing.JApplet {

    static final long serialVersionUID = -1802260290307532227L;

    public void init() {
        initDemo();
    }

    public void start() {

    }

    public void initDemo()
    {
                // This skips the frame window and places the MainPanel directly
                // in the content pane of the Applet
                 Container contentPane = this.getContentPane();
                 JPanel panel1 = new  LESTest();
                 panel1.setPreferredSize( new Dimension(900, 640));
                 this.setSize( new Dimension(900, 640));
                contentPane.add(panel1, BorderLayout.CENTER);
    }
}
```

The program above can be run as-is in an Applet viewer. What you really want to do is to run the program from a web browser. To do that you need to create a jar file of the applet and add the proper initialization code to an HTML page viewable from your web site. Using Eclipse you would select the File | Export option. Select the files and resources in the LESTest directory and any other data files and resources you are using.



In this case the jar file is saved using the name LESTest.jar. This is one of the three jars you will need in order to run the application in a web browser. The other jar files you will need are the qcmatpackjava.jar and qcmatviewjava.jar files found in the \quinn-curtis\java\lib directory. These jar files are uploaded to your server. The code that needs to be added to your web page will look like:

```
<APPLET CODE = "com.quinncurtis.matpackjava.examples.LESTest.Applet1.class"
         CODEBASE = "http://quinn-curtis.com/classes/"
         ARCHIVE = "LESTest.jar, qcmatpackjava.jar, qcmatviewjava.jar"
         WIDTH = 900 HEIGHT = 600>
  This browser does not support Java 1.2 (or greater) applets !
</APPLET> </p>
```

This HTML page (LESTestApplet.htm) can be loaded and the applet run at our web site using the following link:

http://quinn-curtis.com/LESTestApplet1.htm

Note the following:

- The CODE attribute specifies the full namespace of the main applet class, com.quinn-curtis.qcmatpack.examples.LESTest.Applet1.class
- The ARCHIVE attribute specifies two jars, one that contains the code of the applet (LESTest.jar), and the other the QCMatPack libraries (qcmatpackjava.jar and qcmatviewjava.jar).
- The CODEBASE attribute tells the browser in which directory to find the applet. If the applet is in the same directory as the calling HTML page the CODEBASE attribute is not necessary.
- The HEIGHT and WIDTH attribute specify the pixel dimensions of the window the applet will display in

This is the simple version of enabling an applet in a web page. The more complicated way, and the way recommended by Sun is to use the HTML <OBJECT> tag. Apparently the <APPLET> tag has been deprecated by the <OBJECT> tag in the HTML standard and Sun is encouraging web designers to use it. You can read all about it from the source at:
http://docsun.cites.uiuc.edu/sun_docs/C/solaris_9/SUNWaadm/JVPLUGINUG/p5.html

They have a program that will convert <APPLET> code to <OBJECT> code, and forces the browser to download a Java plug-in if one is not already present on the computer. It also uses the EMBED tag instead of OBJECT if the Netscape browser is used. The Netscape trick involves the use of the <COMMENT> tag. Read about that in the Sun article hyperlinked above. The converted version of the HTML applet code looks like.

```
<OBJECT
    classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
    WIDTH = 800 HEIGHT = 600
    codebase="http://java.sun.com/jpi/jinstall-14-win32.cab#Version=1,4,0,mn">
    <PARAM NAME = "code" VALUE =
"com.quinncurtis.matpackjava.examples.LESTest.Applet1.class" >
    <PARAM NAME="codebase" VALUE="http://quinn-curtis.com/classes/" >
    <PARAM NAME = "archive" VALUE =
"LESTest.jar,qcmatpackjava.jar,qcmatviewjava.jar" >
    <PARAM NAME="type" VALUE="application/x-java-applet;version=1.4">
    <PARAM NAME="scriptable" VALUE="false">
    <COMMENT>
    <EMBED
            type="application/x-java-applet;version=1.4"
```

```
        CODE = "com.quinncurtis.matpackjava.examples.LESTest.Applet1.class"
        CODEBASE = "http://quinn-curtis.com/classes/"
        ARCHIVE = "LESTest.jar,qcmatpackjava.jar,qcmatviewjava.jar"
        WIDTH = 800
        HEIGHT = 600
    scriptable=false
    pluginspage="http://java.sun.com/jpi/plugin-install.html">
        <NOEMBED>
    No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
</NOEMBED>
</EMBED>
</COMMENT></OBJECT>
```

This HTML page LESTestApplet2.htm) can be loaded and the applet run at our web site using the following link:

http://quinn-curtis.com/LESTestApplet2.htm

Note that while similar, some of the attributes and tags have a different use. The best place to read about the differences is in the Sun article referenced at http://docsun.cites.uiuc.edu/sun_docs/C/solaris_9/SUNWaadm/JVPLUGINUG/p5.html

The complete source to the HTML pages are found in the LESTest examples directory, file LESTestApplet1.htm and LESTestApplet2.htm).

Note the following:

- The main applet class, com.quinn-curtis.matpackjava.examples.LESTest.Applet1.class is specified using the “code” <PARAM>
- The required jar files are specified using the “archive” <PARAM>.
- The directory that the applet is in is specified using the “codebase” <PARAM>. If the applet is in the same directory as the calling HTML page the “codebase” <PARAM> is not necessary.
- The HEIGHT and WIDTH attribute specify the pixel dimensions of the window the applet will display in
- The rest of the tags and <PARAM> values support the automatic download of the Sun Java plug-in, if the client viewing the applet needs it.

- The <EMBED> section implements an equivalent block of HTML code that is only called if a Netscape browser is used to view the applet.

.

Selected Bibliography

Afifi, A. A. and Clark, Virginia, Computer Aided Multivariate Analysis, Lifetime Learning Publications, 1984.

Atkinson, L. V.and Harley, P. J., An Introduction to Numerical Methods with Pascal, Addison-Wesley Publishing Company, 1983.

Brigham , E. Oran, The Fast Fourier Transform and It's Applications, Prentice-Hall, Englewood Cliffs, 1988.

Chapra, S. C.and Canale, R. P., Numerical Methods for Engineers, McGraw-Hill Book Company, New York, 1985.

Golub, Gene H. and Van Loan, Charles F., Matrix Computations, John Hopkins University Press, 1989.

Harris, F. J., "On the Use of Windows for Harmonic Analysis with the Fast Fourier Transform", *Proceedings of the IEEE*, Volume 66, No. 1, January 1978.

Kleinbaum, David G. et al., Applied Regression Analysis and Other Multivariable Methods, PWS-Kent Publishing Company, 1988.

Lapin, Lawrence L., Statistics for Modern Business Decisions, Harcourt Brace Jovanovich, Inc. 1978.
Rabiner L. R. and Gold, B., Digital Signal Processing, Prentice-Hall, Englewood Cliffs, 1974.

SLATEC Common Mathematical Library, version 4.1 (1993). SLATEC stands for Sandia-Los Alamos-Air Force Weapons Laboratory-Technical-Exchange-Committee, available at http://gams.nist.gov/serve.cgi/Package/SLATEC/.

Smith, James T., C++ For Scientists and Engineers, Intertext Publications McGraw Hill, 1991.

Stoer, J., Bulirsch, R., Introduction to Numerical Analysis, Springer, 1992.

Strang, Gilbert, Linear Algebra and Its Applications, Harcourt Brace Jovanovich, 1988.

Temperature-Electromotive Force Reference Functions and Tables for the Letter-Designated Thermocouple Types Based on the ITS-90. *Natl. Inst. Stand. Technol. Monograph 175*; 1993, http://srdata.nist.gov/its90/main/.

Trefethen, Lloyd N. and Bau, David III, Numerical Linear Algebra, Society for Industrial and Applied Mathematics, 1997.

Wilkinson, J. H and Reinsch, C., Linear Algebra, Springer-Verlag, 1971.

INDEX

Applets, 1, 6, 133, 143
Benchmark timer, 19, 20, 21
BMTimer, 19, 20, 21
C#, viii, 1, 35, 40, 62, 89, 113
Cholesky factorization (Complex), 8, 12, 13, 14, 15, 21, 23, 59, 63, 66, 67
Cholesky factorization (Real), 8, 13, 14, 15, 21, 59, 62, 66, 67
Complex matrix base class, 12, 13, 20, 33, 35, 42, 43, 51, 52, 64, 65, 66, 67, 73, 75, 76, 77, 78, 79, 111, 112, 113, 114, 118
Complex matrix class, vii, 2, 7, 8, 11, 12, 13, 19, 20, 23, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 62, 63, 72, 77, 79, 80, 109, 110, 111, 118, 119
Complex numbers, 2, 4, 7, 8, 11, 12, 13, 20, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 36, 37, 38, 40, 43, 44, 46, 50, 62
Cubic splines, 9, 18, 19, 21, 93, 101, 102, 103
CubicSpines, 9, 18, 19, 21, 93, 101, 102, 103
Curve fitting, 9, 18, 21, 93, 94, 96, 97, 98, 99
CurveFit, 9, 18, 21, 93, 94, 96, 97, 98, 99
DComplex, 2, 4, 7, 8, 11, 12, 13, 20, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 36, 37, 38, 40, 43, 44, 46, 50, 62
DDMat, vii, 2, 7, 8, 10, 12, 13, 19, 20, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 55, 62, 72, 74, 77, 84, 85, 87, 88, 89, 91, 96, 97, 98, 99, 102, 103, 109, 111, 112, 113, 114, 115, 118, 119, 131
Digital signal processing, viii, 6, 7, 9, 19, 21, 105, 107, 108, 109, 110, 111, 112, 113, 114, 115
DMatBase, 12, 13, 20, 33, 35, 42, 43, 51, 52, 64, 65, 66, 67, 72, 73, 75, 76, 83, 86, 90, 96, 111, 113, 118
DMatViewer, 8, 9, 10, 20, 103, 104, 117, 118, 119
DSP, viii, 6, 7, 9, 19, 21, 105, 107, 108, 109, 110, 111, 112, 113, 114, 115
DTriplet, 19, 20, 21, 36, 54
DXMat, vii, 2, 7, 8, 11, 12, 13, 19, 20, 23, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 62, 63, 72, 77, 79, 80, 109, 110, 111, 118, 119
EigenBase, 16, 21, 69, 74
EigenQL, 8, 16, 21, 69, 71, 72, 74
EigenQR, 8, 16, 17, 21, 23, 69, 71, 72, 74, 75, 76
EigenQR2, 8, 12, 16, 17, 21, 23, 69, 71, 72, 74, 76, 77
Eigensystem solver base class (Complex), 8, 12, 16, 17, 21, 23, 69, 71, 72, 77, 78, 79
Eigensystem solver base class (Real), 16, 21, 69, 74
Eigenvalues, 69, 71, 75
Eigenvectors, vii, 71, 72
EigenXBase, 16, 21, 69, 77, 78, 80
EigenXLR, 8, 12, 16, 17, 21, 23, 69, 71, 72, 77, 78, 79
Error processing, 20, 21, 47, 129, 130, 131
Exponential curve fitting, 9, 18, 21, 93, 94, 96, 97, 98, 99
Fast Fourier Transform, viii, 6, 7, 9, 19, 33, 105, 106, 107, 109, 110, 111, 112, 113, 114, 115
Gauss-Jordan (Complex), 8, 12, 13, 14, 15, 21, 23, 59, 63, 64, 67
Gauss-Jordan (Real), 8, 13, 14, 15, 21, 59, 62, 63, 64, 67
Integer vector, 20, 21, 83, 86, 90
IVect, 20, 21, 83, 86, 90
Java Applications, 6, 143
Least squares linear regression, 8, 17, 18, 21, 82, 83, 84, 85, 86, 87, 88, 89
LESBase, 13, 14, 20, 59
LESCholesky, 8, 13, 14, 15, 21, 59, 62, 66, 67
LESGJ, 8, 13, 14, 15, 21, 59, 62, 63, 64, 67
LESLU, 8, 13, 14, 20, 59, 62, 64, 65
LESQR, 8, 13, 14, 15, 16, 21, 59, 62, 65, 66
LESXBase, 13, 14, 21, 59
LESXCholesky, 8, 12, 13, 14, 15, 21, 23, 59, 63, 66, 67
LESXGJ, 8, 12, 13, 14, 15, 21, 23, 59, 63, 64, 67
LESXLU, 8, 12, 13, 14, 21, 23, 59, 63, 64, 65
LESXQR, 8, 12, 13, 14, 16, 21, 23, 59, 63, 65, 66
Linear equation solver base class (complex), 13, 14, 21, 59
Linear equation solver base class (Real), 13, 14, 20, 59
LU decomposition (Complex), 8, 12, 13, 14, 21, 23, 59, 63, 64, 65
LU decomposition (Real), 8, 13, 14, 20, 59, 62, 64, 65
MatBase, 12, 20, 33
MatError, 20, 21, 47, 129, 130, 131
Matrix base class, 12, 20, 33
Matrix viewer (complex, 8, 9, 11, 20, 117, 118
Matrix viewer (real), 8, 9, 10, 20, 103, 104, 117, 118, 119
MatSupport, 20, 21
MulReg, 8, 17, 18, 21, 82, 83, 84, 85, 86, 87, 88, 89
Multiple regression, 8, 17, 18, 21, 82, 83, 84, 85, 86, 87, 88, 89
NIST ITS-90, viii, 6, 9, 121, 122, 123
Polynomial curve fitting, 9, 18, 21, 93, 94, 96, 97, 98, 99
QL eigenvalue/eigenvector algorithm, 8, 16, 21, 69, 71, 72, 74
QR eigenvalue/eigenvector algorithm, 8, 16, 17, 21, 23, 69, 71, 72, 74, 75, 76, 77, 78, 80
QR factorization (Complex), 8, 12, 13, 14, 16, 21, 23, 59, 63, 65, 66
QR factorization (Real), 8, 13, 14, 15, 16, 21, 59, 62, 65, 66
QR2 eigenvalue/eigenvector algorithm, 8, 12, 16, 17, 21, 23, 69, 71, 72, 74, 76, 77
Rational fraction curve fitting, 9, 18, 21, 93, 94, 96, 97, 98, 99
Real matrix base class, 12, 13, 20, 33, 35, 42, 43, 51, 52, 64, 65, 66, 67, 72, 73, 75, 76, 83, 86, 90, 96, 111, 113, 118

Real matrix class, vii, 2, 7, 8, 10, 12, 13, 19, 20, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 55, 62, 72, 74, 77, 84, 85, 87, 88, 89, 91, 96, 97, 98, 99, 102, 103, 109, 111, 112, 113, 114, 115, 118, 119, 131
Regression statistics, 20, 21, 84, 85, 87, 88, 89, 91, 96, 97, 98, 99
RegStats, 20, 21, 84, 85, 87, 88, 89, 91, 96, 97, 98, 99
Split regression, 8, 17, 18, 21, 90, 91
SplitMulReg, 8, 17, 18, 21, 90, 91
Stepwise regression, 8, 17, 18, 21, 85, 86, 87, 88, 89
StepwiseMulReg, 8, 17, 18, 21, 85, 86, 87, 88, 89
TCLin, 21, 121, 124, 125, 126, 127
Thermocouple linearization, 21, 121, 124, 125, 126, 127
XMatBase, 12, 13, 20, 33, 35, 42, 43, 51, 52, 64, 65, 66, 67, 73, 75, 76, 77, 78, 79, 111, 112, 113, 114, 118
XMatViewer, 8, 9, 11, 20, 117, 118
XTriplet, 19, 20, 21, 36